THE UNIVERSITY OF HULL

Specification and use of component failure patterns

being a Thesis submitted for the Degree of Doctor of Philosophy

in the University of Hull

by

Ian Philip Wolforth BSc(Hons) MSc

May 2010

Abstract

Safety-critical systems are typically assessed for their adherence to specified safety properties. They are studied down to the component-level to identify root causes of any hazardous failures. Most recent work with model-based safety analysis has focused on improving system modelling techniques and the algorithms used for automatic analyses of failure models. However, few developments have been made to improve the scope of reusable analysis elements within these techniques. The failure behaviour of components in these techniques is typically specified in such a way that limits the applicability of such specifications across applications. The thesis argues that allowing more general expressions of failure behaviour, identifiable patterns of failure behaviour for use within safety analyses could be specified and reused across systems and applications where the conditions that allow such reuse are present.

This thesis presents a novel Generalised Failure Language (GFL) for the specification and use of component failure patterns. Current model-based safety analysis methods are investigated to examine the scope and the limits of achievable reuse within their analyses. One method, HiP-HOPS, is extended to demonstrate the application of GFL and the use of component failure patterns in the context of automated safety analysis. A managed approach to performing reuse is developed alongside the GFL to create a method for more concise and efficient safety analysis. The method is then applied to a simplified fuel supply and a vehicle braking system, as well as on a set of legacy models that have previously been analysed using classical HiP-HOPS. The proposed GFL method is finally compared against the classical HiP-HOPS, and in the light of this study the benefits and limitations of this approach are discussed in the conclusions.

Publications

Some of the work presented in this thesis has already been published in conference proceedings and journal papers. An earlier revision of the Generalised Failure Language can be found in (Wolforth *et al*., 2008). An extended version of this paper including the brake-by-wire case study from Chapter 4 can be found in (Wolforth *et al.*, 2010b). The current version of the language developed in this thesis, with additional details of language semantics, can be found in (Wolforth *et al.*, 2010a).

Acknowledgements

Many have inspired me to get this far, but I could not have made it without all the help and support I received from various people, not forgetting those who made the journey that much easier thanks to their friendship.

First I thank my family for making this possible, thanks to their support over the many years I have been at university, especially the time spent living at home while I wrote (and re-wrote) my thesis. Thanks to Mum, Dad and Grandad Joe for putting up with me during my grumpy periods. Easily the best thing to have happened during these years was the arrival of my nephew and niece, Philip and Eleanor, so extra thanks go to my brother David and his wife Jo.

I also want to thank Graham Asbury for being a good friend all these years, and hopefully for many more.

Next I want to thank and extend my best wishes to all my friends and colleagues at Hull who have since left or are still there, particularly Nidhal Mahmud, Amer Dheedan, Shawulu Nggada and Septa Sharvia.

Many thanks go to Lars Grunske for his help with papers, especially his excellent work on developing the semantic rules for the GFL.

Huge thanks also to David Parker for his work on HiP-HOPS and his other ‘projects’.

Massive thanks to Martin Walker, for being a friend and for working with me on the development and extensive re-development of the GFL. He also did a terrific job working on and editing the papers we managed to get published together, so thanks again for everything.

Finally, I will always be grateful for the opportunity offered to me by Dr Yiannis Papadopoulos to undertake postgraduate research. Only with his advice, guidance, inspiration and seemingly endless patience over the years, have I managed to write this thesis. Considering where I started from all those years ago, I struggle to explain to anyone reading this just how influential his supervision has been. I have been extremely lucky, as I very much doubt I would have got this far or achieved so much having been mentored by anyone else.

Contents

Figures

Tables

1 Introduction

1.1 Safety critical systems

The modern world is increasingly filled with technology to provide tools and automation for a wide variety of tasks. This diversity ranges from small scale embedded devices to very large scale industrial sites. As people become more dependent on technology, the safety and reliability of systems becomes paramount. Reliability is the ability of a system to operate uninterrupted by failures, while the concept of system safety is the ability of a system to avoid causing hazards to people and the environment. The term *safety critical system* refers to those systems whose operation or failure causes such hazards (Leveson, 1995).

The nature of many safety critical systems generally requires that some element of human interaction or control is in place, rather than complete automation of every function. For the foreseeable future, some level of manual input will still be required for these kinds of systems as complete automation in many cases is simply not trusted. Certain legal requirements will apply to some extent with regards to safe operation for the protection of the people who interact with the system, but there is also the economic factor in the simple desire to have systems work reliably over the course of their operational life, with minimal ‘downtime’ for any repair or modification which would otherwise affect productivity or provision of service. The increasing reliance on technology combined with the costs and potential risks of failure mean that safety and reliability become increasingly more important factors in system design.

Safety critical systems are checked for their safety and reliability properties before they are deployed for operation. The process of *safety analysis*, the assessment of these criteria, begins during the design of the system and continues throughout the lifecycle.

A *safety case* (Wilson *et al.*, 1997) for instance is incrementally built in step with the design and must be kept updated so far as the system is eventually decommissioned. Any changes to the system during its life will mean the safety case must be re-evaluated to assess the effects and implications of change, thus the safety analysis process is continual. Indeed, results of a safety assessment may necessitate further design changes if for example potential hazards are found.

The aim of the safety analysis is to identify potential hazards associated with the operation or failure of a system and to explore the events that cause such hazards within the architecture of the system and in the environment. Hazardous states typically arise from system *failures*, i.e. conditions in which correct system function is not delivered (Avižienis *et al.*, 2001). Failures can occur at the component level, the low-level constituent parts of a system, or at a system-wide level. Component level failures can propagate or combine with other, less severe failures to cause potentially hazardous system level failures.

Thus the basis for predicting safety comes from identifying potential system failures, relating them to potential causes, and determining their likelihood from the likelihood of those causes.

According to the standard terminology in this field of research, an *error* is the difference between the expected and actual system state that can lead to a failure, with the cause of an error ultimately being a *fault* (Avižienis *et al.*, 2001). There are many types of faults, but generally a fault can arise from internal or external factors, for example a simple electronic component may malfunction over time due to wear, or from interference received from another source. Even the most rigorous design processes cannot always guarantee that a fault will not occur within a system, thus any system of any construction can experience failure at some point during its operational lifetime.

Faults can be *random*, being unexpected events brought about by a physical cause (e.g. wear or corrosion of components), but can be predicted using statistical data gathered from testing or prior knowledge from previous use. Alternatively faults can be *systematic*, a problem with the design or construction of the system (which could be introduced at any point in the lifecycle) meaning, once discovered, the fault occurs predictably during operation.

Prediction of safety properties is now common and indeed mandatory for safety critical systems such as aircraft and nuclear power systems (Vesely *et al.*, 1981). In the case of nuclear power systems for instance, electricity is generated from a highly radioactive source, direct exposure to which would be fatal to people. In the instance of a critical failure in the control system, lives are potentially put at risk and due to this potential hazard, safety features are incorporated into the system so that in the event of control failure, fail-safe and backup systems come online to prevent the escalation of the fault to a hazardous failure. Of course this adds complexity to the system, and these fail-safe features must themselves be checked for safety properties.

In a well designed and constructed system, root causes of hazardous failures should have a very low probability of occurring during its operation. Component faults can be tolerated in certain circumstances when they do not contribute directly to hazards, otherwise they must have very low probability of occurrence or measures must be taken to stop the propagation of errors caused by such faults. Reliability statistics such as 'mean-time-to-failure' are often provided by component manufacturers, so that system builders can be advised on when a component is expected to have reached the end of its safe operational lifespan. These figures can range from thousands of hours for the mechanical components in a typical domestic car, to only a few hours in a motorsport racing car where components operate in extreme conditions. This type of data is essential when analysing a system for failures; when a potential hazard is found, the root causes are checked for their probability of occurrence. Knowing the likelihood of occurrence will influence decisions about the cost-effectiveness of implementing improved safety features for avoiding hazards. Sufficiently unlikely hazards do not demand design iterations while likely hazards with severe consequences necessitate redesign, the aim of which should be either to remove the hazard if possible, or to decrease the likelihood of the hazard for example via incorporation of fault tolerant architecture that can prevent the direct propagation of the component faults that cause the hazard.

To improve efficiency and cost, modern systems rely increasingly on electronics and computers (Ferguson, 2007). This can result in new types of failure occurring, which can compromise safety and complicate safety analysis. For example, a hydraulic control for a vehicle braking system can only ever allow braking pressure to be applied to wheels when pressure is applied to the brake pedal. If the hydraulic system were replaced with an electronic control, braking control signals would be sent to actuators to apply the braking pressure. Now this new system is potentially susceptible to erroneous control signals, possibly caused by interference to the electronics, leading to undesired application of the brakes. Conversely, undesired application of braking pressure is impossible with a hydraulic system.

To summarise, there exists a class of systems that are considered *safety critical*, whose requirements for operational safety must be checked before they can be used by people. No system is free from faults, so they are analysed for potential hazardous failures which could arise from the occurrence of specific faults, or combinations of faults. Quantitative data is used to supplement this analysis to determine the likelihood of failures. Finally, design changes can be made in response to analysis results in order to eliminate or mitigate the effects of a hazardous failure. This complete process is known as *safety analysis*.

1.2 Safety Analysis

Modern safety analysis has been heavily influenced by the nuclear power industry which is forced to deal with a high level of complexity and stringent legal requirements of operational safety. However, the challenge of identifying hazards remains, as does the consequence of making design changes to reduce the risk and the severity of the hazardous states.

Many techniques for safety analysis have been developed, the earliest being tabular methods of compiling information about potential failures such as Failure Mode and Effects Analysis (FMEA) (IEC 60812, 2006) and Hazard and Operability Studies (HAZOP) (Kletz, 1997). These are generally now supplemented by powerful graphical methods such as Fault Tree and Event Tree Analysis (FTA, ETA) (IEC 61025, 1990) which are founded on logic and probabilistic analyses. For example a fault tree diagram is a graphical means to display the root causes and propagation of failures, showing how basic events can combine and propagate to cause a hazard (Vesely *et al.*, 1981). The success of these methods has meant they are now widely used outside of the nuclear industry, such as in the automotive (Papadopoulos and Grante, 2005) and aerospace (Vesely *et al.*, 2002) industries.

Safety analysis is a field that has seen a lot of research and development since its conception, but the core aspects have not changed considerably over time. A common approach, used for example in fault tree analysis, begins with the identification of potential system failures that have a hazardous consequence. A specific failure will be chosen for detailed investigation and is traced back through the system to its root causes, which will typically be component level faults; this is known as a *deductive* approach. Some other techniques work in the opposite way, an *inductive* approach, where effects are determined from causes. For example, FMEA begins with tabulating the potential failures that can be experienced and then examines their effects on the system.

Safety analysis is traditionally a manual, arduous task. Analyses are performed by a team of specialist analysts on a system implementation. If safety and reliability aspects are deemed inadequate, then costly redesigns may be essential. Furthermore, subject to the effect on the system of such a change, analyses may then need to be performed again. A complex system design can make the process even more difficult, and as a result several software tools have been developed to help with analyses. The common aim with these is to automate as much of the process as possible, which is often the focus in the development of contemporary safety analysis methods.

In theory, the safety analysis process should start as early as possible so that checking the system for potential failures occurs at early design stages. This way, problems can be identified earlier in the development lifecycle, meaning unnecessary redesign cost could be reduced. The difficulty is that without the complete knowledge of component interactions at each stage when analysis is applied, this approach is not possible. In order to analyse for safety at design stages, a *model* of the complete system is required. With appropriate software tools, models can be mechanically interpreted, enabling a form of model-based safety analysis (MBSA).

The model that provides the basis of analysis may be architectural and provide the topology of the system as a diagram showing the interconnected components, each assigned information about their function and failure behaviour. It can also be a representation of behaviour and show transitions of components and the system from normal to degraded and failed states.

An example of a model-based analysis technique is HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies), a method and tool that performs a semi-automatic safety analysis on system designs (Papadopoulos *et al.*, 2001) that are represented as architectural topologies augmented with failure data. Using a model-based approach, components in the system hierarchy can be annotated with logical expressions of failure behaviour. In HiP-HOPS, these component-level annotations are interpreted by the tool and in combination with the topology of the system, used to automatically synthesise fault trees and FMEAs for the model.

1.2.1 Reuse in designs and safety analysis

Managing complexity in engineering projects is a problem in itself, especially as more functionality is demanded from contemporary systems. To ease development, there is a tendency to reuse solutions that have been successful in one project to aid the development of another. This can take the form of reusing particular hardware components or subsystems, or even software where there is a computer-controlled element. This has varying degrees of success due to reuse being susceptible to inappropriate application. For instance, a problem could be identified by an engineer as being ‘similar enough’ to one seen in a previous system that a prior solution is deemed adequate. In this situation, the *ad hoc* ‘cut & paste’ approach to reuse may lead to an unsatisfactory implementation, with costly consequences. A more appropriate method is to control and manage reuse through a systematic process (Kelly and McDermid, 1998). Being able to appropriately reuse a solution means the design process can be completed more quickly, bringing time and cost saving benefits. In addition, by applying well-known solutions to problems, the result is a design which can be considered more ‘trusted’, which is a very desirable factor when developing safety critical systems.

In the pursuit of reusing as much design knowledge as possible, there are established concepts to manage the various levels and type of reuse. The most common forms of reusable design elements are *components*, *frameworks* and *patterns* (Johnson, 1997). Each of these provides a certain level of abstraction, allowing reuse across various applications. Components are self-contained 'blocks' of well-known design, frameworks provide a reusable basis to build upon, and patterns define generic behaviour which can be adapted to context. Each of these is used within the field of engineering and can be very useful concepts to aid and speed up design and development of new systems. The concept of patterns can be found for example within software engineering as *design patterns* (Gamma *et al*., 1995), and is established as a technique that can enable robust, consistent design of common software solutions. They promote good engineering practice and a certain degree of reusability, for example in class design and interaction. Design patterns take the form of documentation showing an implementation of the solution, plus additional information listed for guiding its application; the documentation as a whole becomes the 'pattern'. In another context, the notion of patterns can be used to guide the construction of arguments in a safety case (Kelly and McDermid, 1997), in this case the pattern provides an abstract, reusable guide to building the arguments, with flow diagrams and explanatory text, together constituting the 'pattern'. Not all aspects of a design can be reused of course, nor would such blanket reuse be desirable, for many systems require bespoke components. However, attempting to further extend the scope and ease of reusability is a worthwhile cause, as it undoubtedly yields an improvement in productivity (Kehren *et al*., 2004). Knowing that reuse is an important feature of modern system design, including safety critical design, leads to the question: can reusable design elements, particularly patterns, be brought into safety analysis methods, allowing some degree of reuse of information about safety properties?

Contemporary MBSA methods, for example HiP-HOPS and other methods examined in this thesis, do enable some reuse of safety analyses. These tend to base their reuse on the *compositional* nature of the systems they analyse. Systems in general are developed via decomposition into more manageable parts, and can be represented in design as hierarchies of subsystems and basic components. Reflecting this compositional architecture, many modern safety analysis methods rely on component failure models to predict system failure. These component models can in theory be reused across applications in the analyses of systems that incorporate the corresponding components. While such reusability is highly desirable, the expected level of reuse that compositional analyses should bring is currently very difficult to achieve. The limiting factor is that this kind of reuse depends on the fact that the reused components have not been modified in any way which affects their operation and thus their safety properties. However, even for simple components this may not be possible due to *context changes*; information about specific failures may become irrelevant within a new application. An example of this can be seen in an automotive brake-by-wire system. Here, the components have a specific application context; communication buses for example carry signals that relate to activation of the brakes or the amount of pressure to be applied. In most MBSA approaches, the component models that define the failure behaviour of a component like a bus in a given application make application-specific references to the errors propagated through the bus, in this example, deviations of braking messages. This is despite the fact that a bus component can be designed to propagate signals without needing to know the meaning or context. In summary, while a communication bus can work in various application contexts, information about its failure behaviour maintains its context specificity and cannot necessarily be reused along with the physical component. Thus one of the major obstacles to achieving greater reusability within compositional model-based safety analyses is this context-specific nature; in particular the context-specific definitions of component failure behaviour.

There are cases where a component will behave consistently and independently of its system context, giving an opportunity to investigate context-independent failure behaviour. In a rudimentary example of this, a pipe enables the flow of material without needing to know what is flowing. A total blockage of the pipe will always result in a loss of output flow, again regardless of the material. With regards to electronic components, they too can exhibit this type of context-independent behaviour. An electronic data bus designed to propagate signals can experience the same type of failure, omission of output signals, regardless of the system and meaning of the signals.

There are many more cases where context-independent behaviour is expressed by individual components. When such context-independent behaviour exists, it can be identified as a reusable element, *a pattern of behaviour* that holds in any application. If capture and reuse of such patterns were possible, then the safety analysis of complex systems could also benefit from reusing well-established and thoroughly studied component failure behaviour. Not only would this save time, money and effort, but it would potentially improve the dependability of designs by incorporating aspects which are proven to be effective and are essentially ‘trusted’ elements. The difficulty however in capturing such patterns lies in the current inability to define generalised failure behaviour in most techniques that define the state-of-the-art in model-based safety analysis.

1.3 Research hypothesis and Objectives

The hypothesis put forward and tested in this thesis is that:

> "More efficient safety analyses can be achieved via a new approach to *generalising descriptions of failure behaviour*. This approach will allow in practice the more concise specification and limited reuse of *patterns of failure behaviour*"

The above hypothesis raises two research questions that the thesis attempts to address:

1. How can descriptions of failure behaviour be generalised?
2. To what extent can patterns of behaviour be used within safety analysis?

Testing the hypothesis requires meeting several objectives. The first of these is to examine relevant MBSA techniques to evaluate the current extent of achievable reuse. There are several MBSA methods which aim to promote reuse in safety analysis and which, therefore, need to be examined in order to inform the objectives and direction of this work.

The second objective is to develop a concept for generalised annotation and demonstrate that the proposed concept works by extending an existing safety analysis method to enable specification and reuse of patterns. The candidate method for this extension, HiP-HOPS, is an established tool and already provides some support for reuse. Systems can be hierarchically decomposed within HiP-HOPS to allow component level descriptions of failure, but the annotations themselves are generally restricted to the particular context of the system. This immediately limits the scope of reusability of failure annotations within HiP-HOPS. Here the aim is to achieve an extension of the existing component annotation syntax with language features that enable specific context to be eliminated from annotations. The result is an annotation language that enables more generalised descriptions of component failure behaviour (Wolforth *et al.*, 2010a).

The final objective in this thesis is to validate the proposed method. Case studies provide an opportunity to show how the method is applied, and to evaluate the contribution brought by the new approach. In the study included in this thesis, several example models are analysed to compare the extended HiP-HOPS method that includes capabilities for patterns with the classical HiP-HOPS. These are relatively simple models based on prototype systems provided by industry, comprising various types of components; mechanical, electrical and programmable. In the course of these case studies, equivalent specifications are developed in classical and extended HiP-HOPS and the results are compared. The use of patterns is demonstrated to lead to shorter specifications that can be reused within the same application.

To quantify the effectiveness of the new approach, data mining techniques are used to identify patterns of failure behaviour in legacy safety analyses that have been carried out using classical HiP-HOPS in a number of industrial case studies. It is shown that the number of component level annotations can be significantly reduced via the proposed extensions to HiP-HOPS, and that it is possible to capture patterns of behaviour which can then be reused within the same model. This data forms the basis of the evaluation of the pattern method; the amount of reuse that can be expected and the effectiveness of the reuse.

To summarise, this thesis makes the case for the development and application of patterns of failure behaviour in model-based safety analysis. The improvement to the efficiency of analyses is investigated, as is the overall effectiveness of the approach. To support the argument put forward, the thesis:

1. Develops a language extension for HiP-HOPS to enable patterns of behaviour to be specified

2. Implements tool support that enables analyses of component failure patterns and derivation of specific instances of failure behaviour from such patterns

3. Develops a managed approach for the application of patterns and investigates the potential for reuse

4. Evaluates the feasibility of patterns on a series of case studies

5. Compares and contrasts the new method with the classical HiP-HOPS process

1.3.1 Structure of the thesis

The thesis consists of three main chapters describing the background, theory, and validation of the concept being argued. Chapter 2 presents a review of available literature on classical and contemporary safety analysis techniques, with an evaluation of reuse attainable in safety analysis and more generally in the wider field of engineering. Chapter 3 sets out the concept for patterns and proposes the linguistic constructs necessary for enabling generalised failure descriptions. Chapter 4 discusses the application of patterns and how they can be managed to enable greater levels and more appropriate reuse. A series of case studies are performed, the results of which are used for validating the new technique. Chapter 5 concludes the thesis, evaluating the contributions of the work along with the limitations, and finally discusses further work.

2 Background

This chapter reviews available, relevant literature in the field of safety analysis to provide a more detailed view of the research problem. The first part explores the methods used in safety analysis; the original, *classical* techniques which are still in use today, through to the *contemporary* methods developed to deal with the more complex designs which are typical among modern systems. The techniques are discussed and evaluated, with the emphasis on their benefits and shortcomings with respect to the reusability within their processes, such as the scope, ease and potential for reuse. The focus here is on the qualitative aspects of safety analysis; studies of probability are essential in rigorous analyses, but this work only considers extension to the core qualitative process which is primarily concerned with identification and establishment of relationships between causes and effects of failures.

The second part of this chapter discusses the concept and scope of reusability in general, as found in the wider field of engineering, with a focus on those techniques applied in software development. The aim is to highlight which aspects can or could be transferred into and applied within safety analysis. In conclusion, this study of available literature highlights the important aspects to consider in developing a contemporary approach to reuse in safety analysis.

2.1 Classical Safety Analysis

There are many approaches and techniques used for safety analysis, varying from relatively simple data tabulation, through to highly complex, computerised simulations of functional prototypes. This is largely in response to the variation of system designs; the variety of complex systems in today's industries means one specific analysis technique is simply not sufficient (Kaiser *et al.*, 2003). As a result, new techniques are developed to meet the demands of contemporary system designs, and it is notable that a significant amount of these tend to be variations of a small number of classical analysis methods (Gould *et al.*, 2000).

It is often the case that the basic principles of the classical methods are still valid in modern approaches, but some limitations inherent in the original methods also still hold in the newer ones. It is important to discuss these original methods so to observe the evolution of the more recent techniques by highlighting the successful features from the classical methods that have been carried over into contemporary approaches.

2.1.1 Failure Mode and Effects Analysis

Failure Mode and Effects Analysis (FMEA) (IEC 60812, 2006) is a widely used, tabular process for checking the safety and reliability properties of a system. FMEA and its variants are used across a large range of industries, and this level of adoption has occurred despite the difficulties encountered when using this classical method.

An FMEA is typically a manual, labour intensive process for finding the effects of component failures on a system. For each component in the system and for each failure mode of that component, analysts investigate the effects on the system and record results in an FMEA table. The linear, tabulated form of the information gives a straightforward representation, making it simple for manual inspection of results and identification of critical component failure modes that cause severe effects with high probability. Variants of FMEA tables contain different attributes including contributing factors, failure probability or failure rate, detectability, controllability and other parameters of interest. For example Failure Mode, Effects, and Criticality Analysis (FMECA) (ARP 926, 1967) is a commonly used extension of FMEA which includes the criticality of failures in the assessment, for analysing their probability and severity.

Table 1 gives an example of a row in an FMEA of a vehicle braking system. Here the brake pedal can experience the failure mode ‘No signal’, leading to the effect ‘Omission of braking’. The detail of the cause and its failure rate (probability per unit time) are given along with the severity, giving clear information so that the analyst’s attention can be focused towards those failure modes deemed most critical. Because the FMEA process requires a lot of detail about a system before it can be performed, it is mainly done when all details of the design are known.

Component	Failure mode	Failure cause	Failure effect	Severity	Failure rate
Pedal	No signal	Pedal failure	Omission of braking	Catastrophic	$5e^{-5}$
...	...	...	...	...	...

Table 1 Example FMEA table

Like other classical methods, traditional FMEA suffers from scalability problems. The first of which is that an FMEA table only describes failures that have a single cause. Multiple-cause failures, which are not uncommon in today’s complex systems, are not considered in the analysis process. The reason for this, and the root of the second problem, is that the consideration of every possible combination of component failure modes can lead to extremely large tables. Even relatively simple systems with a large number of failure modes can suffer this problem during the analysis, e.g. in a system with 100 component failure modes, there are approximately 5,000 combinations of two of those failure modes to be considered. Clearly, the manual interpretation of such large volumes of data, often ranging into hundreds of thousands of combinations of failures, is not feasible within the time or budget allowed for such analyses.

The traditional FMEA process is capable of finding multiple-cause failures, but they are generally not considered within a study because they are too numerous. Being a manual process, the effort required for a complete multiple-cause FMEA in most cases is too great. However, recent developments to FMEA production (Parker *et al.*, 2006) have enabled automation and improved tool support which have made feasible the discovery of multiple-cause failures. In this approach, multiple failure mode FMEAs are constructed from a set of automatically constructed fault trees. Given the deductive nature of fault tree analysis, is it possible in this approach to generate multiple failure mode FMEAs without exhaustive enumeration and evaluation of all possible combinations of component failure modes. Price and Taylor (2002) have also developed an approach to FMEA synthesis via fault simulation in which results of automated FMEAs may be 'pruned' to extract combinations which are deemed significant.

Potentially, some data from an FMEA can be reused, though effects must be carefully checked. If an FMEA only contains simple failure modes, it may be the case that a particular component would exhibit the same standard behaviour regardless of context, and if operating conditions are identical, also likely to maintain the same failure rate. The obstacle to achieving more reuse is the informal, manual nature of the analysis; descriptions of failures (the entries within a table) are subject to the engineer's consistency with the formatting. Without a systematic approach to developing consistent FMEA tables, efforts to reuse information across analyses have limited success.

2.1.2 Hazard and operability studies

Hazard and operability studies (HAZOP) (Kletz, 1997) was one of the earliest methods created for checking system safety and is still widely used, especially in the chemical industry. It aims to be a generic technique to find causes and effects of hazards, meaning it is both a deductive and inductive method. These are specified as deviations from the original design intent; for example the flow of material through pipes can deviate from its normal operation as a result of ‘no flow’ (a blockage), which could ultimately lead to a hazardous state. This method been adopted throughout safety-critical industries with very little change to the original method, due to its generalised format and overall relative ease of use.

A HAZOP study is performed by manually searching for hazards in a systematic way and tabulating the results. The basis is to use identifying *keywords* to describe potential hazards and then to derive their potential causes. This keyword concept was an attempt to allow some form of reuse within safety checking. Giving classifications to failures, a common keyword, allows the analyst to make general statements about failure types. For example, the following keywords are valid in a HAZOP study and are typical of the types of failures considered within other types of safety analysis:

OMISSION or *COMMISSION*: e.g. the absence or presence of an expected message

HIGH or *LOW*: e.g. an out-of-range signal

EARLY or *LATE*: e.g. a message has arrived before or after it was expected

The result of a HAZOP study is a table bringing together all the gathered information on the system relating hazardous failures to their cause(s). Table 2 gives a partial HAZOP table; the example shown is of a generic system where a signal node fails to deliver an output, leading to a system wide failure. The resolution to this hazard is to replicate the critical node to provide some backup redundancy in event of node failure.

HAZOP was created and is useful for identifying specific failures which can occur in large scale industrial sites. For modern computer controlled systems, it is very difficult to manage a HAZOP study with the high numbers of potential component failures. The tabular format of the results can make it simple to quickly identify the causes of hazards, but with a large table it becomes less practical, for instance trying to find the information for a particular hazard among hundreds of table entries. Furthermore, classical HAZOP studies only determine single-cause hazards, making it unsuitable for thorough analyses that must consider all possible hazards.

Keyword	Deviation	Cause(s)	Consequence(s)	Resolution
OMISSION	No signal	Node failure	System failure	Replicated node
...	...	...	...	...

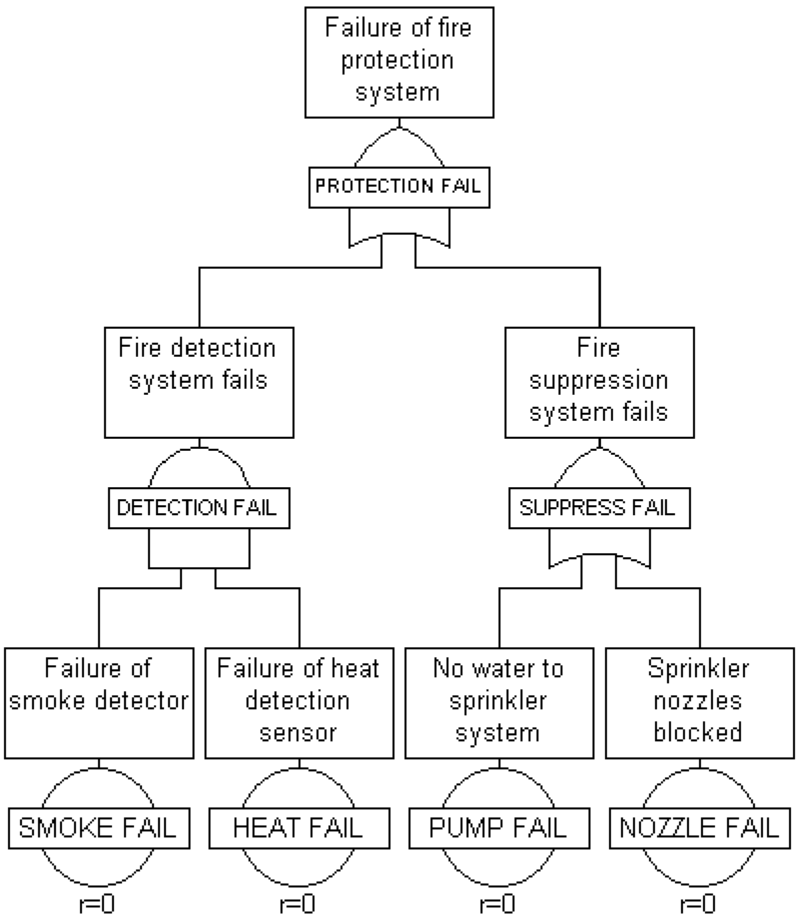
Table 2 Example HAZOP table

Hazard analyses are performed throughout the life of a system (Redmill *et al.*, 1997). Early studies can take place when design detail of components and interconnections is available. A preliminary hazard analysis (Vesely *et al.*, 1981) can provide useful information that can be carried over to more detailed analyses that take place at later stages when more detailed designs are available. HAZOP studies then continue throughout the lifecycle of a safety critical system, as it is necessary to reassess potential hazards whenever operating parameters or conditions change. Thus any scope for employing reusable elements across HAZOP studies would be beneficial.Smith and Harrison (2002) have investigated techniques to enable reuse of descriptive arguments for hazard classification. A row in a HAZOP table can form an argument based on the consequence and resolution columns; the basis for the reuse lies in the similarity in argument structure. Reuse candidates are identified by matching keywords or consequences from previously defined arguments. Case studies have shown a ‘considerable amount’ of achievable reuse, but a difficulty still remains in determining the suitability of arguments for a particular context.

2.1.3 Fault Tree Analysis

The problems encountered in classical FMEA motivated the development of fault tree analysis (FTA) (Vesely *et al.*, 1981), currently one of the most prominent safety analysis techniques. FTA was first developed in 1961 by Bell Laboratories, in connection with a U.S. Air Force contract, to investigate the potential causes of an inadvertent missile launch. Once the results were seen, fault tree analysis was recognised by Boeing as being a significant system safety analysis tool, eventually being used on the design and evaluation of commercial aircraft. Following the aerospace industry, the nuclear power industry began using fault tree analysis in the design of nuclear power stations. It was the nuclear power industry which is responsible for the development of many fault tree analysis tools. In 1967, NASA hired Boeing to undertake a comprehensive fault tree analysis on the entire Apollo system, following the Apollo 1 launch pad fire. The Challenger space shuttle disaster in 1986 resulted in a safety evaluation of the main engines, which highlighted the applied benefits of fault tree analysis. Today, fault tree analysis is widely recognised as a robust and useful process which helps to develop insight into the failure behaviour of a system.

A fault tree diagram represents the propagation of failures through a hardware or software system in terms of subsystem or component failures. These 'basic events' are linked into a tree structure using Boolean logic, representing diagrammatically the logical relation between events. The paths through the fault tree from bottom to top determine which of these combinations of basic events cause the top event. The Fault Tree Handbook (Vesely *et al.*, 1981) is widely accepted as the definition of the standards for fault tree notation and method. An example fault tree is given in Figure 1.



Failure of fire protection system
PROTECTION FAIL
Fire detection system fails
Fire suppression system fails
DETECTION FAIL
SUPPRESS FAIL
Failure of smoke detector
Failure of heat detection sensor
No water to sprinkler system
Sprinkler nozzles blocked
SMOKE FAIL
HEAT FAIL
PUMP FAIL
NOZZLE FAIL
r=0
r=0
r=0
r=0

Figure 1 A small fault tree (Andrews, 1998)

Fault tree analysis (FTA) is a technique that can be applied to a fault tree to determine:

- all the necessary and sufficient combinations of basic events that cause the top event of the fault tree (qualitative analysis)
- the probability of occurrence of these combinations and that of the top event (quantitative analysis)

Combinations of basic events that cause the top event (i.e. system failure) are known as cut sets. By identifying critical design elements, cut sets help to focus the design. For example, the design can then be improved by incorporating redundant components that can automatically replace critical components when they fail.

An interesting and useful property of a fault tree is that it is essentially a graphical representation of a complex Boolean expression, meaning the whole or sub-trees can be expressed textually, which also extends to the cut sets. A cut set can be considered then as a written expression relating a failure to its causes; from the Figure 1 example:

Protection Fail = Smoke Fail AND Heat Fail OR Pump Fail OR Nozzle Fail

Much of the recent innovation into the fault tree method has come from the analysis side; various algorithms for the automatic extraction of minimal cut sets exist (Wolforth, 2005), and research continues into finding faster and more efficient methods. In addition, fault tree notation is being continually improved and extended so that more detailed representation can be made within a diagram, allowing fault trees to remain a powerful tool today. For instance, traditional fault trees cannot accurately model dynamic behaviour, where the order of events is critical, so many attempts have been made to extend the notation for creating temporal fault trees (Walker, 2009).

Because fault trees only give a system-wide view of failure, one difficulty with using the method is that any redesign made to a system, even small changes, may require that the entire fault tree is reconstructed and reanalysed. Also cut sets describe very specific events within a particular system, so reuse of cut sets i.e. analysis results is not feasible across designs.

Dehlinger and Lutz have shown that reuse of fault trees within a ‘family’ of systems is possible. A product line (or family) is a set of systems that are all developed from a core specification, where members of the product line differ only in a small number of allowed variables. The reuse benefits from an engineering perspective (shared design elements, architecture, etc. within the family) can also be related to safety analysis through modelling extensions to classical fault tree analysis (Lutz *et al*., 1998). In product line fault tree analysis (PLFTA), design variables are associated with the tree’s leaf nodes. An analysis can then derive a fault tree for each product as the variables change. A semi-automated tool ‘PLFaultCAT’ has been developed for PLFTA, however the manual effort of the analysts and engineers, as with classical FTA, is still the main factor in the accuracy and completeness of analyses and therefore the extent of reusability (Dehlinger and Lutz, 2006).

2.1.4 The Safety Case

A higher level approach for arguing convincingly that a system will safely perform its operations is the concept of the safety case (Wilson *et al.*, 1997). The purpose is to give a documented assurance that system safety requirements are met. Indeed the safety case is a legal requirement in many regulated industries in order to provide such assurances to a regulating body. The safety case report which argues the case is a set of complex documents, manually constructed by a team of analysts, which summaries results of safety assessments.

The exact constituents of a safety case document vary with application and are not subject to a common specification (Wilson *et al.*, 1997), but a safety case must at least include clear, well-defined arguments, supported by evidence, that a system will be acceptably safe throughout its life (Wilson *et al.*, 1995). These are the key elements of the safety case; an argument without supporting evidence is unconvincing, and evidence without an argument makes it difficult to distinguish how safety requirements have been met. Below is an example of a safety argument from (Kelly and Weaver, 2004):

The Defence in Depth principle (P65) has been addressed in this system through the provision of the following:

- *Multiple physical barriers between hazard source and the environment (see Section X)*
- *A protection system to prevent breach of these barriers and to mitigate the effects of a barrier being breached (see Section Y)*

The evidence for an argument may be for example fault trees or FMEAs for the system. Because safety case evidence is often based on the results of prior safety analyses (Wilson *et al.*, 1997), a safety case is not itself a means to discover failure behaviour, rather it is a useful means to present safety information in a structured manner.

Developing the safety case is no easy task, the most pertinent difficulty being their potential size, in terms of their construction, management and subsequent analysis, particularly for large studies. This can lead to difficulties in maintaining a complete document, necessitating a significant manual effort. Addressing these problems, recent improvements to the construction and management of safety cases have been made in the form of software tool support with the Safety Argument Manager (SAM) (McDermid, 1994) and the Goal Structuring Notation (GSN) (Kelly, 1998). SAM is a tool for performing various classical safety analyses, which provide the results (evidence) for use in the subsequent safety case development. The GSN, as part of SAM, is used to help set out the claims of the arguments, aiding the most difficult part of developing the safety case. Following the goals gives a progression through the argument, typically formed as a hierarchy of lower level arguments.

Because of the complexity involved in their production, individuals responsible for creating the document will often take elements of previous safety cases to help their development (Kelly and McDermid, 1997). The danger is that successful, convincing arguments are taken as ‘trusted’ and can be reused without thorough investigation of the applicability (Kelly and McDermid, 1998). This kind of reuse is based on the manual observation that two elements are similar enough to be used in another design, with few changes needed. Clearly when this is handled manually by people, errors and inconsistencies can easily be introduced through improper reuse. Despite the difficulties, skilled safety engineers can successfully produce rigorous safety cases to give the evidence that even a complex system is safe to operate.

2.2 Contemporary Safety Analysis

Modern systems bring new challenges to safety analysis. These are caused by the large scale (many components, e.g. small sensors and micro switches) and increasing complexity (e.g. programmable/software components introducing new types of failures) of these systems.

Clearly the manual nature of classical techniques poses problems. It makes it increasingly difficult to achieve a complete analysis within the time and budget constraints of most projects. The lack of mechanisms for safe reuse of analyses is another problem. For example, subsystems and components may be reused to perform the same or similar function across applications. However, with the lack of a proper framework for reuse, it cannot be assumed that their respective safety analysis can be safely reused across applications. Part of the problem is the separation of classical safety analysis from the design process, which makes it difficult to trace the effects of design modifications in the analysis.

This section reviews several modern safety analysis techniques that attempt to address some of these problems. Most of these techniques integrate design with analysis and derive system safety analyses from a model of the system. Some of these techniques follow a compositional approach in which system safety analyses are constructed from component safety analyses via a process of composition. This enables a greater scope for reuse of analysis elements within their methods. Many of these techniques have their foundations in the classical methods, often making improvements over the original and solving some of their drawbacks highlighted in the previous discussion.

2.2.2 Model-Based Safety Analysis

Many contemporary approaches use a *model* of a system and its behaviour to improve and automate (to a degree) some aspects of the safety analysis. Generally, a model is a representation of a system describing architecture and/or functional behaviour. Interconnected components in an architectural model also determine the flow of failures through a system, which some methods exploit to create a corresponding failure logic model (Lisagor *et al.*, 2006).

Many tools exist to help develop models; some rely on text-based descriptions whereas others allow a model to be developed visually, for example in MATLAB/Simulink (Mathworks, 2009). When a system is under design, a model can be produced with varying levels of detail. For example early design may only yield a relatively simple model consisting of blocks of interconnected components. As more detail is provided, for example when functions are allocated to components, more detailed models can be produced. Models can be updated throughout the lifecycle to reflect function or architecture changes in the system. The implication for safety analyses is that effects of design changes can be more quickly assessed. As models are defined using formal and semi-formal languages and notations, they can be mechanically interpreted (parsed) and therefore used as the basis for automated analysis. This type of automation, while being able to speed up the overall process, has the potential to reduce the opportunity for manual errors in the analysis. This is a major benefit and one of the main motivations for new model-based analysis methods which provide as much automation as possible. It is important to note however that none of the contemporary methods reviewed in this chapter provide fully automated analyses; some manual effort is still required, mainly at early stages of their process where the modelling must be done.

2.2.3 FPTN

Failure Propagation and Transformation Notation (FPTN) (Fenelon and McDermid, 1993) was developed to allow a compositional approach to safety analysis and has influenced the development of subsequent methods. The motivation for its development was to solve some of the problems of the classical methods while also creating a more integrated approach to safety analysis (Fenelon *et al.*, 1994). FPTN was primarily developed to aid fault tree analysis and FMECA when applied to software, but can be applied to any system with a compositional architecture. A component failure can be traced through the system to determine its effects (inductively) to create an FMECA. Alternatively an output failure can be traced to its root causes (deductively) to create a fault tree for analysis. The method aims to give a generalised view of system failure behaviour and provide a link between such deductive and inductive analyses.

An FPTN module directly represents an architectural component in a model. A complete diagram will show how failures flow through the system through the interconnected components (modules), thus creating the failure model. Each module may then contain a number of sub-modules to create a component hierarchy (Grunske and Neumann, 2002). The result is that both deductive and inductive analyses can be performed with an FPTN diagram, as failure modes can be traced backwards to discover causes, or forwards to discover effects.

Each module contains the name of the component, a criticality level and where applicable, any recovery mechanisms (‘exception handlers’). These are included within the header of a module, as shown in the example module in Figure 2. Each module also contains the failure modes of the component (internal failures), expressions relating input and output failure modes, and may be modelled such that modules are nested hierarchically. SHARD-style classifiers (Ezhilchelvan and Shrivastava, 1989) are used for definition of failure types, i.e. timing (t), value (v), etc.

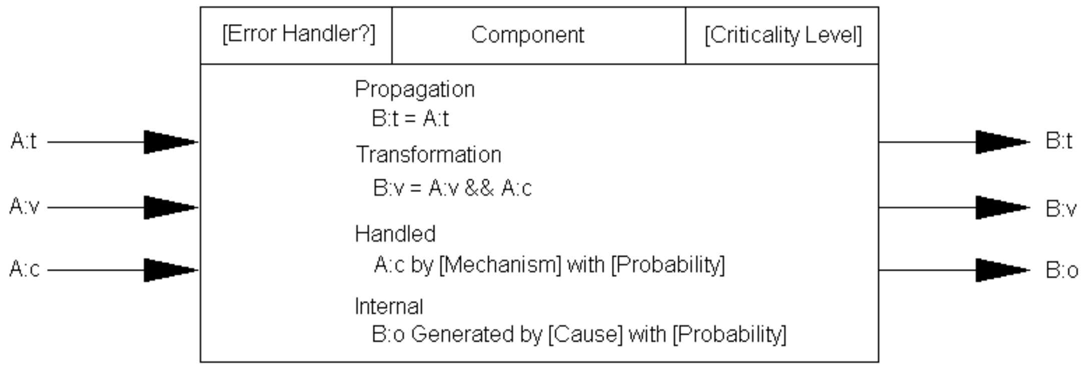


Figure 2 Example FPTN module (Fenelon *et al.*, 1993)

It is possible to reproduce the failure expressions contained in a module in a simplified 'sum-of-products' form, which are equivalent to the minimal cut sets generated in FTA, where the output failure mode is considered as the top event. This means that an individual FPTN module can be considered an abstraction of the set of fault trees for the component represented by the module.

The main benefit of FPTN is the end-to-end modelling, linking deductive (e.g. FTA) and inductive (e.g. FMECA) approaches. As the modelling reflects the system architecture, faults can be seen to propagate through components, arguably making root cause identification easier. The drawbacks of FPTN though, are that it is strictly a descriptive notation and cannot be used to analyse failure propagation (Wallace, 2005), and has struggled to receive tool support (Paige *et al.*, 2009). Also an FPTN module represents only known failure propagation between components, so connections between modules are not made unless a known failure propagates between architectural components. Finally, with regards to reusability, a key problem with the FPTN method is that the detail of the modules is context specific; they need to be reconstructed whenever a change takes place in a component, because a change may alter the type of failure propagated and also to where the failure is propagated. This limits their reusability despite the compositionality of the diagram, but recent improvements to the notation (discussed in the following section) have allowed more generalisation and improved reusability.

2.2.4 FPTC

Influenced by FPTN, FPTC (Fault Propagation and Transformation Calculus) is a similar method to its predecessor for compositional analysis of systems (Wallace, 2005). A significant contribution of this approach is the enhanced syntax for the system failure modelling. The FPTC notation adds useful abstractions and generalisations, for example a wildcard character can be included to indicate any type of failure, rather than listing every type individually. Variables can be used, which can also represent any type, but their main purpose is to indicate propagation within an expression:

$$(\,late\,,\ \_\ ) \rightarrow (\,low\,)$$

$$(\,omission\,,\ f\ ) \rightarrow (\ f\ )$$

This example set of expressions would be used to define the behaviour of a single component. The first expression describes that a *late* failure on the first input and any (specified as a wildcard, '_') failure on the second input causes a *low* failure on the output, and the second expression describes that a failure *f* is propagated to the output if there is an *omission* of the first input and failure *f* on the second input. The use of wildcards and variables make it possible to create generic expressions, a very powerful abstraction which achieves a greater level of reusability, certainly improved over FPTN.

The FPTC method requires an architectural model of the system (example in Figure 3), into which the components and connections have an FPTC expression assigned to them to describe behaviour. FPTC is designed for the domain of real-time software and requires that architectural descriptions are in the form of graphs, consisting of nodes that represent hardware and arcs that represent communication protocols. One available notation for developing such graphs is the Real-Time Networks formalism (Paynter *et al.*, 2000). Once annotated, automated analysis can proceed with the system acting as a token-passing network, where tokens representing the possible failures flow from one component to another (Paige *et al.*, 2009). Propagation and transformation of failure tokens happens at the components or connections in the model. Expressions are 'run' using the tokens, and the system continues to run until the tokens no longer change. Failure information can thus be obtained by studying the tokens at output or specific points in the system. Recent work further extends the syntax of FPTC to allow probabilistic analysis of failures (Ge *et al.*, 2009). This adds a probability value to each expression so that calculated probabilities are passed through the model in addition to the failure tokens.

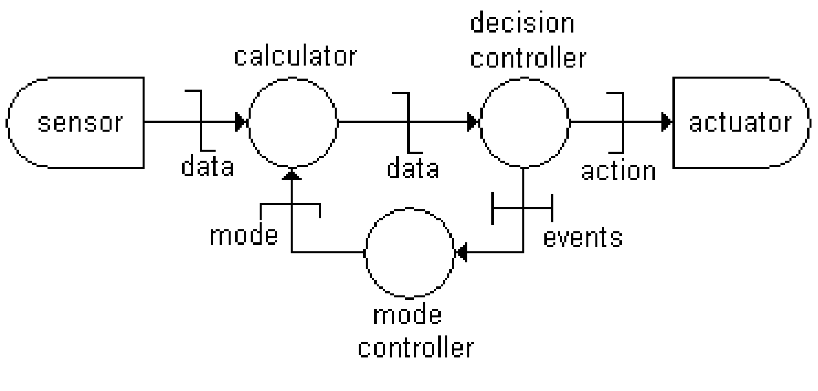


Figure 3 Example Real-Time Networks architecture (Wallace, 2005)

The development of FPTC addresses some key deficiencies in its predecessor FPTN, such as the requirement to reconstruct modules due to architecture changes and the context-specific nature of the failure expressions. However a difficulty is encountered with the propagation algorithm of FPTC; the algorithm is inductive (forward analysis from causes to effects) and can suffer effects of combinatorial explosion when assessing combinations of failures (Wolforth *et al.*, 2010a).

2.2.5 HiP-HOPS

The automation of safety analysis can be very beneficial, especially considering the potential for elimination of any human errors in traditionally manual processes, such as fault tree construction. HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) is a tool that performs a semi-automatic safety analysis on system designs (Papadopoulos *et al.*, 2001). It is capable of synthesising fault trees and FMEAs from a system model which can then be used for further study within HiP-HOPS. These steps are fully automated, but a manual effort remains in the initial system modelling and failure annotation.

In the course of the design life-cycle, HiP-HOPS can operate with progressively more detailed models as they are produced and refined. Thus opportunities can arise to reuse information gained from earlier analyses and enables a consistent and continuous assessment of the system as the model evolves. At the earliest stages, such a model may be an abstract block diagram which shows the composition of the system from a purely functional point of view, for instance input and output transactions among functions, which can be decomposed into lower level sub-functions. When functions are allocated to hardware components at a later stage in the development, the model will then be a representation of the physical architecture of the system in its entirety. Each refinement of the model that increases the available information about the system will yield more detailed analysis results from HiP-HOPS.

HiP-HOPS requires that potential component failures are first identified using a variation of the classic HAZOP technique. This initial study is used to describe component failure behaviour as a set of Boolean expressions relating output failures to their causes, such as internal malfunctions or input failures. Failures are defined by the analyst into classifications, typically following HAZOP types (omission, commission, etc.), but can be more specific if required. In the form of the expression, a class of failure occurring on a *port* (component input or output connections) is known as a *deviation*, for example:

```
Omission-output = Omission-input OR Internal_malfunction
```

The above expression defines a component failure where an omission of the input, or an internal malfunction of the component, causes an omission of the output. Sets of these expressions of failure behaviour are manually annotated to components in the model; tool support has been developed to enable this within commercial modelling software[1]. Once components have been annotated, their failure behaviour in combination with the topology of the system (from the model) is used to determine the failure propagations through the system, and used to automatically construct a set of fault trees for the system. Following this, the fault trees are automatically analysed, resulting in a table of minimal cut sets. Such a table is analogous to a multiple failure mode FMEA as it shows direct relationships between component and system failures (Papadopoulos *et al.*, 2004). The complete HiP-HOPS process can be seen in Figure 4.

[1] The HiP-HOPS tool has been shown to work with EAST-ADL2 (Chen *et al.*, 2008), MATLAB/Simulink (Papadopoulos and Maruhn, 2001), and Simulation X (Uhlig *et al.*, 2007), and is currently being harmonised with AADL (Feiler and Rugina, 2007).

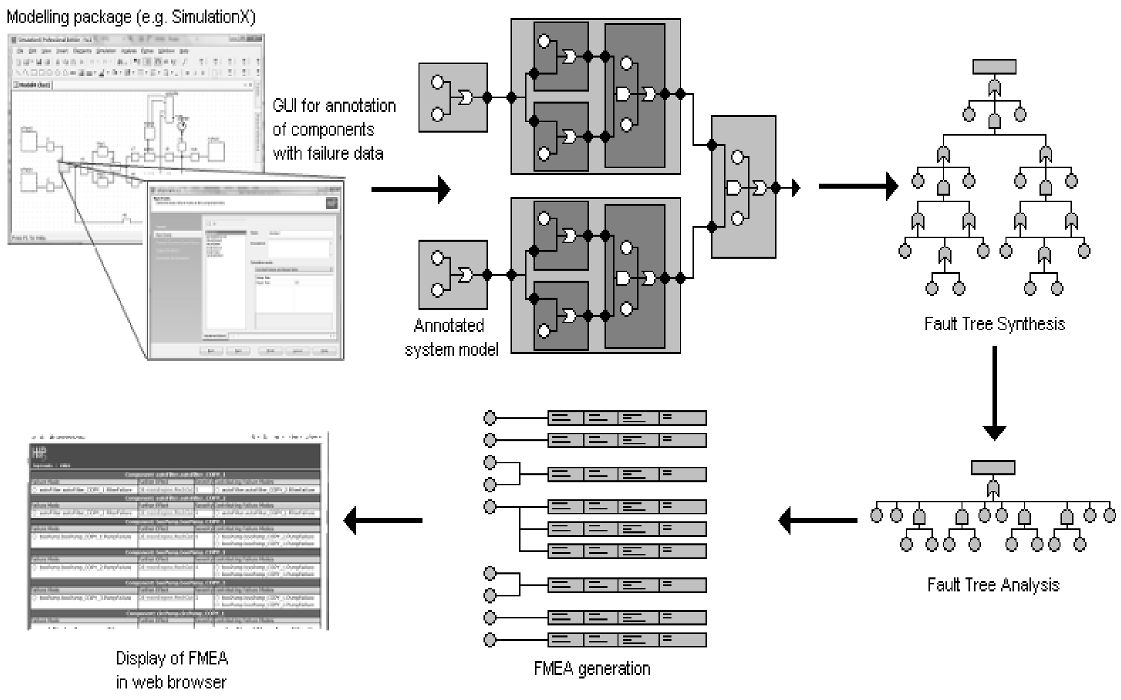

Modelling package (e.g. SimulationX)
GUI for annotation of components with failure data
Annotated system model
Fault Tree Synthesis
Fault Tree Analysis
FMEA generation
Display of FMEA in web browser

Figure 4 The HiP-HOPS process

The effectiveness of HiP-HOPS depends upon the component annotations; it is essential that appropriate failure data is used. As the only manual step, it is also susceptible to error. Reusability within HiP-HOPS is limited by the context-specific component failure behaviour expressions, though if a component expresses a general behaviour and is annotated thus, the expression could be reused safely along with the component.

2.2.6 Component Fault Trees

Though methods like HiP-HOPS give a model of the failure behaviour of a system represented as a set of fault trees, in the pursuit of compositional analysis, modelling systems at the component-level cannot be done accurately with traditional fault trees. Fault trees can be modularised (Wolforth, 2005), but these modules do not necessarily represent architectural components. The reason for this is that fault trees model paths of failure propagations to a root hazard, rather than the system architecture itself. Independent sub-trees can be extracted from a fault tree and defined as modules (specifically sub-trees which contain no repeated events), however components are often influenced by other components and therefore cannot be modelled as an independent entity in this way.

Component Fault Trees (CFTs) (Kaiser *et al.*, 2003) are an extension to the traditional fault tree concept to allow compositional modelling and analysis. The extension is to the fault tree notation, so that component ports can be included within the diagram. Otherwise, they are the same as traditional fault trees and can have the same qualitative and quantitative analysis methods applied to them. Using the new notation, componentised fault trees are linked by input and output ports, represented as a solid triangle, as shown in Figure 5. Basic events are now stored in their own components as independent events (even if they have the same name), which manages to avoid the 'repeated event' issues from traditional fault tree analysis.

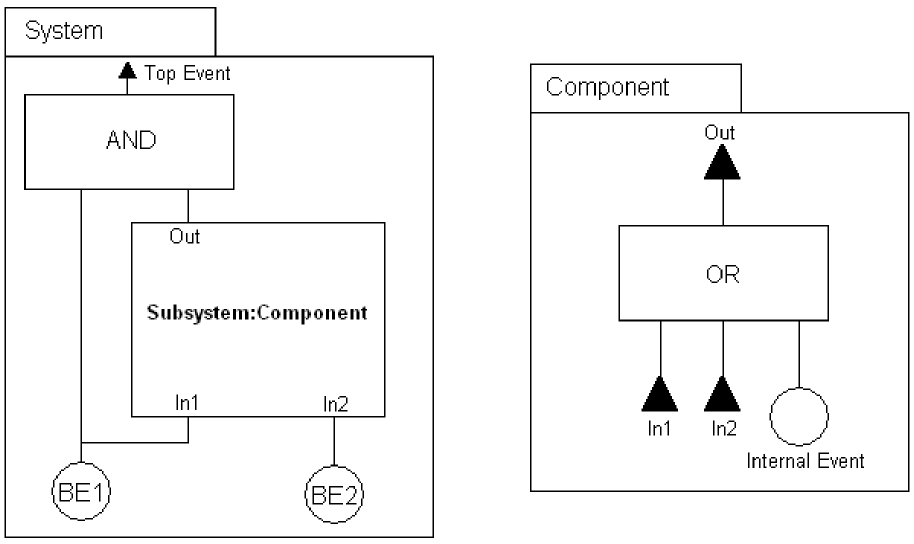


Figure 5 An example CFT

The fault trees used in CFT are in fact a generalised form of standard fault trees (Mäckel and Rothfelder, 2001) called Cause Effect Graphs (CEGs). These allow the common-cause failures (repeated events) to only need a single representation, and they also allow multiple top events. CEGs make CFTs smaller, clearer and easier to analyse, which are significant benefits particularly when modelling larger systems. In (Heilmann *et al.*, 2007), CFTs are used in an industrial case, analysing a railway braking system. In this report, the automatic generation of CFTs helped to manage complexity arising from a large number of combinations of subsystems.

As components are independent, their behaviour is encapsulated and they can be developed separately, assuming the necessary ports for fault tree components have been determined. These features come together to create a reusable component concept for fault trees and analysis, with the added advantage that the decomposition is from system to sub-components, which is generally more intuitive than from top event to basic events.

2.2.7 State-Event Fault Trees

Traditional fault trees do not have the ability to model the temporal order of events, i.e. sequences of events or states and transitions; they only model static behaviour. State-Event Fault Trees (SEFTs) (Kaiser and Gramlich, 2004) are an evolution of CFTs and allow the modelling of dynamic failure behaviour[2]. The ability to represent system states and state transitions within a fault tree leads to a much more in-depth and accurate analysis. This is particularly important for systems with a real-time element, where the traditional combinatorial fault tree is not suitable for modelling such dynamic behaviour. It allows a true representation of the actual functioning of a typical system, for example the sequence of events and their effect on the failure behaviour. The difficulty however lies in capturing, representing and analysing the SEFTs. An example SEFT is shown in Figure 6.

[2] Modelling of dynamic behaviour and associated temporal analysis is beyond the scope of this thesis, nevertheless it is worth mentioning as an example of contemporary safety analysis.

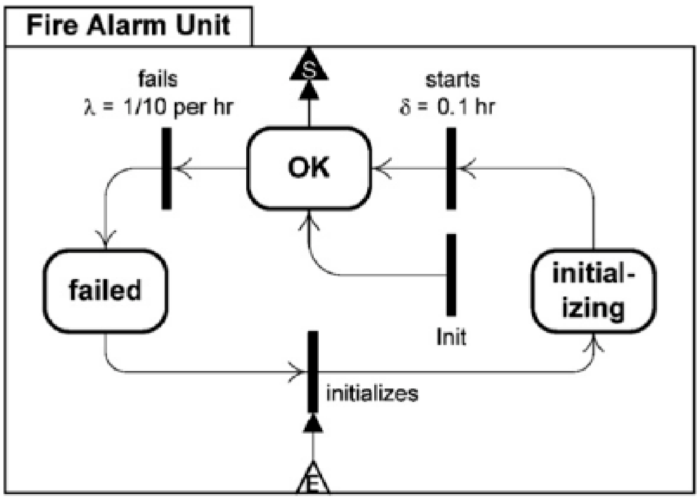


Figure 6 Example SEFT from (Kaiser *et al.*, 2007)

SEFTs extend the standard fault tree notation and add graphical elements for the representation of states and events, and also temporal and causal edges to the graph; an important distinction here is that states and events are not necessarily failures of a component. This extended notation also includes and extends the port concept from component fault trees, where they are now refined into state ports and event ports. Therefore the level of achievable reuse is similar to that found in CFTs.

Traditional fault trees can be analysed with various methods by manipulation of the underlying Boolean expression structure. However SEFTs are only analysed by first transforming the tree into Deterministic and Stochastic Petri Nets (DSPNs) (Ciardo and Lindermann, 1993). Currently, further analysis of the DSPNs is performed by external tools e.g. TimeNET (German and Mitzlaff, 1995), but work is being done on the integration and automation of these methods (Kaiser *et al.*, 2007). The detail required for analysis using SEFTs mean that the technique is susceptible to state-space explosion problems and so performance suffers (Grunske *et al.*, 2005). This problem can be addressed to some extent by a dual-analysis technique, whereby the static parts of the system are analysed with combinatorial FTA algorithms and the dynamic parts analysed by the more powerful methods mentioned previously. The effectiveness of this approach is however highly dependent on the type of system under analysis.

2.2.8 Safety Case Patterns and composable safety cases

The traditional safety case is a document that argues and presents evidence for the safety of a system, however when the system is large and complex, the safety case is generally large and complex also. Managing changes to the safety case throughout the life of a system can be challenging, with changes to regulatory requirements, additional safety evidence and changing designs all affecting the original argument (Kelly and McDermid, 2001). Successful arguments from prior studies are often reused in order to simplify the construction of the safety case, but problems can arise with its management when an *ad hoc*, ‘cut & paste’ approach has been employed in the development (Kelly and McDermid, 1998). To address these problems, the concept of Safety Case Patterns (Kelly and McDermid, 1997) has been developed to ease the production of safety cases by using a more abstract approach to the general construction principles. Along with a graphical notation to help guide the development of safety cases, these extensions to the method aim to promote reuse and resolve some issues of the manual construction process.

Safety Case Patterns (example Figure 7) are essentially templates for the construction of safety case arguments, given in the GSN format, with added documentation. The overall format is based on design patterns (Gamma *et al.*, 1995) with some adaptation for the description. For instance, included in the documentation is applicability information for describing the circumstances under which the pattern should be applied. Also sample text is given for the safety case under development, similar to example source code in design patterns.

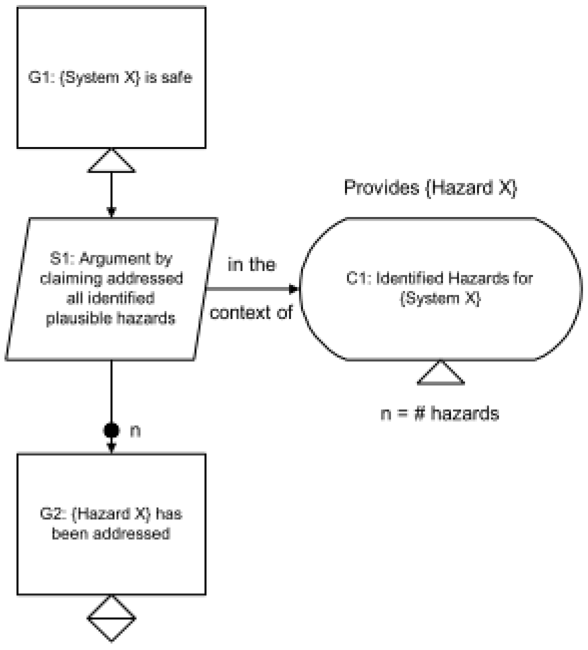

G1: {System X} is safe
Provides {Hazard X}
S1: Argument by claiming addressed all identified plausible hazards
in the context of
C1: Identified Hazards for {System X}
n = # hazards
n
G2: {Hazard X} has been addressed

Figure 7 Hazard Avoidance Pattern from (Kelly and McDermid, 1997)

Safety Case Patterns can certainly enable a degree of reuse and aid in the development of safety cases, but despite these additions, safety case construction is still a laborious manual process.

To achieve a greater benefit from the compositional design of safety critical systems, a compositional approach to safety case development is required (Kelly, 2001). The GSN has been extended further with elements that enable the modular construction and representation of safety cases. The interfaces between modules are the crucial element for composable safety cases. A safety case module interface, as specified by (Kelly, 2001), must contain:

1. Objectives addressed by the module
2. Evidence presented within the module
3. Context defined within the module
4. Arguments requiring support from other modules

Inter-module dependencies:

5. Reliance on objectives addressed elsewhere
6. Reliance on evidence presented elsewhere
7. Reliance on context defined elsewhere

These elements are required for an interface to be properly defined and thus to ensure the composed argument will be consistent, especially when modules are reused across applications. The principal benefit to modularising a safety case is managing change, i.e. maintainability. With a compositional approach, changes can be isolated to a set of modules, as opposed to traditional safety cases where changes may have an impact that necessitates the safety case be reassessed. Clearly, Safety Case Patterns and composable safety cases address some of the main difficulties with safety case maintenance, and achieve an improved and managed approach to reuse.

2.2.9 AADL

The Architecture Analysis and Design Language (AADL) (Feiler *et al.*, 2006) forms the core of what is an extensive tool for the specification and analysis of dependable systems. It was originally developed for formally describing real-time embedded avionic systems, thus it includes modern features such as the capability for hierarchical modelling of software and hardware components and their interactions. It is an industry standard for avionics and other embedded real-time systems. The language is extendable, and one such extension which is of particular interest is the Error Model Annex (Feiler and Rugina, 2007). This provides a sub-language for the creation of error models, describing errors and failure behaviours of components, which are then associated with components in an architectural model. The intention of the error model annex is to provide support for qualitative and quantitative analysis of dependability attributes (Rugina, 2005).

An AADL model can represent hierarchies of components or subsystems, and so an error model's representation also extends from the system to component level. An error model will describe the state changes a component will experience when presented with a particular fault, either from an internal source or propagated from another component. An example of an AADL Error Model is given in Figure 8.

```
error model Example1
features
ErrorFree:  initial error state;
Failed: error state;
Fail, Repair: error event;
CorruptedData: out error propagation
      {Occurrence => fixed 0.8};
end Example1;

error model implementation Example1.basic
transitions
ErrorFree-[Fail]->Failed;
Failed-[out CorruptedData]->Failed;
Failed-[Repair]->ErrorFree;
properties
Occurrence => poisson 1.0e-3 applies to Fault;
Occurrence => poisson 1.0e-4 applies to Repair;
end Example1.basic;
```

Figure 8 Example AADL Error Model (Feiler and Rugina, 2007)

When a system's architectural model is combined with the system's error model, automated analysis can be performed by one of several available methods, or by other AADL extensions. These include, for example, fault tree generation (Joshi *et al.*, 2007) and Generalised Stochastic Petri Nets (Rugina *et al.*, 2007), the latter used for modelling dynamic or state-based behaviour.

A useful feature is that error models can be stored in a library, which provides a repository for the component association and thus provides a degree of managed reuse. There are two types of reusable error model; the basic model, which defines component error states and how they change due to error events and propagations, and the derived model which defines the error state of a component in terms of the error states of its subcomponents. The error models that are reusable, as far as the annex allows, still have architecture dependencies due to component-specific properties in the model definition. The onus then remains on the analyst to ensure that error models are reused appropriately.

2.2.10 Formal Verification for Safety Analysis

The contemporary methods discussed so far are similar in that they are all capable of compositional safety analyses. There is however another class of methods used for safety analysis that must be discussed; these are formal verification techniques which support safety analysis. Like other compositional methods, these use a model as input, but can perform an analysis with simpler information about failures; only local failure modes are required, with propagation of failures inferred from the model.

Formal verification techniques are widely used and provide a greater degree of automation, but are not without their drawbacks. Generally, formal verification techniques are inductive, working from cause to effect. In contrast to a deductive approach, this can lead to combinatorial explosion when combinations of failure modes are analysed. The processing requirements for these methods are also far more computationally expensive due to the extensive checking that must be performed. Nevertheless, these methods are valid and have their place among the contemporary safety analysis techniques (Lisagor *et al.*, 2006).

One such method, simulation, automatically derives failure information about a system from the algorithmic execution of a functional model. The aim is to find out how the system may perform if it were operating as designed, providing information about reliability, availability and maintainability. The simulation process may require a considerable amount of time, particularly as a model must be subjected to repeated executions in order to gain reliable data. The multiple runs can generate large sets of results, but the main drawback here is that potentially not all failures are found even with multiple executions (Price and Taylor, 2002).

Another common method used in industry for formal verification is model checking. The basic principle of the process requires that a system and its properties are expressed as a mathematical model and are checked for meeting safety requirements through computational algorithms. Model-checking can be used to automatically prove safety properties at any stage of a model's development. A general view of the process is given in Figure 9. The 'model' here refers to a representation of system requirements (what the system does) or design (how it does it), which along with the system properties (the properties to be checked, i.e. safety properties that must be 'true') are used as input to a model-checking tool. The result of the evaluation outputs positive if the model satisfies these properties, or if they are not satisfied, a counter-example is produced by the tool. Through detailed examination of the counter-example, problems can be pin-pointed and a model can be manually corrected so that the system properties will be satisfied on repeated application of the model-checker.

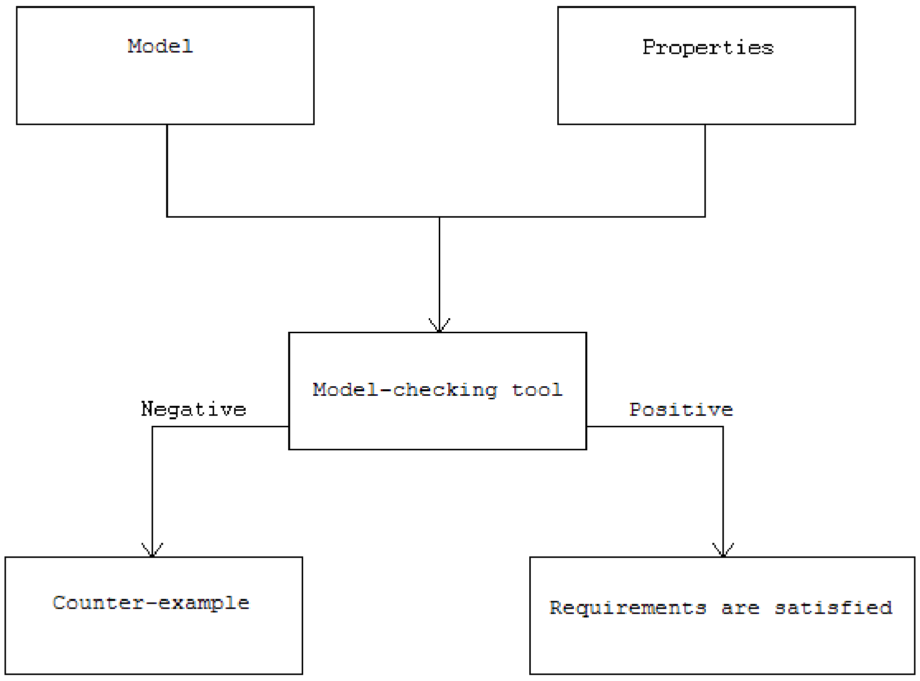


Figure 9 Process of model-checking

Model-checking can be useful in safety analyses, one such ability being the formal checking of fault trees (Thums and Schellhorn, 2003). Thums and Schellhorn define fault tree semantics which formalise fault tree events; logic gates in a tree are assigned a ‘correctness’ and a ‘completeness’. Correctness is a guarantee that sub-events lead to the top event and completeness is a guarantee that only those sub-events lead to the top event, i.e. no causes have been missed during the analysis. Any omissions in the tree are found as a counter-example by the model checker. In another example, model-checking was used very successfully on NASA’s DEEP SPACE 1 system (Havelund *et al.*, 1999), finding several previously unknown errors.

FSAP/NuSMV (Formal Safety Analysis Platform, Symbolic Model Verifier) (Bozzano and Villafiorita, 2003), for example, is a tool for automating safety analyses which combines several techniques. It can perform automatic fault tree generation and model verification and aims to provide a platform that can be used throughout the development of a safety critical system from its design through to the assessment of its safety properties. The platform requires a formally-written model of the system for validating the functional requirements, which can be extended to include component failure modes for safety analyses. A useful feature is that generic safety requirements are stored in a library included with the tool, from which predefined failure modes can be included into a model.

A key drawback of model-checking techniques are that output results require manual verification, due to the fact that 'false negatives' can occur with an incorrect input model or specification (Clarke *et al.*, 1999), e.g. a mistake in the input can lead to the model checker producing a negative result based upon the erroneous input. Thus the result has to be verified against the input to determine if it is a valid counter-example.

Another approach which is based upon model checking is failure injection. In this method, failure modes are added to a model in the same manner as a component (hence 'injected'). When a model is executed/run, a failure mode can be active or not, with the model checker controlling the activation of the failure modes, for example if all failure modes are deactivated, the model will exhibit its normal behaviour. The model checker will produce a list of failure modes that violate a requirement, which is similar to a list of minimal cut sets produced from FTA.

Formal verification provides 'true' automated analysis (Lisagor *et al.*, 2006), and provides an implicit guarantee of 'correctness' of analysis results. Early lifecycle models can be used by these techniques however the problem faced by such methods is the potential complexity of input models, resulting in a computationally expensive process.

2.2.11 Comparison of classical and contemporary safety analysis

Classical safety analysis methods were developed primarily to support analysts working in the aerospace and nuclear industries. The techniques have found lasting success by providing straightforward methods that yield useful results for safety assessment.

Some of the earliest methods, for example FMEA and its variants, are still usefully employed for hazard identification and relating causes of failure to their effects. Fault tree analysis improves on the inductive tabular methods by providing a deductive analysis of failure combination and propagation throughout a system. This and the subsequent analysis of fault trees provide effective means for identification and analysis of root causes of hazardous failures and their likelihood. FTA has become one of the most widely used safety analysis methods today.

Methods that are simpler to perform generally yield less detailed results, but as safety critical systems have tended to become far larger and more complex over time, the classical methods have encountered problems when faced with the increased complexity found in modern systems. The general difficulties shared by all the classical methods are:

- the *manual production* of results and their potential to be of great volume; such large sets of data make the manual interpretation and adequate management of results very difficult. This also makes errors more likely in very large analyses.
- the *maintainability* of analyses; when changes are made to a system, analyses have to be performed again, or results reconstructed, either way resulting in substantial extra effort.

Another issue is the time required and high cost of analyses; due to these factors, a safety analysis is typically only performed after a design is finalised. Although this will yield the required safety information about the final system, it leads to further increases in costs and development time when problems are found and redesigns are necessary. As complexity inevitably continues to increase, the difficulty in applying classical methods increases further (Papadopoulos *et al.*, 2001).

The contemporary approaches bring into safety analysis improvements, extensions and new techniques developed to address those drawbacks of the classical methods. The modern compositional approach to safety analysis began in the mid-1990s. Since then, continuing developments have been made resulting in several different methods, each tackling the problems of managing complexity in their own way. Compositionality is a common feature; with the trend for complex systems to have a composable design incorporating reusable, off-the-shelf components, compositional analysis seems to be a reasonable way of rationalising complex safety assessments. As a result, contemporary methods are capable of performing analyses at the component or subsystem level and aim to provide some reuse via their compositionality.

It is common for compositional safety analysis methods to make use of an early architectural design model of a system to perform their analyses, which saves time, effort and cost. A major benefit over the classical methods is this allows a degree of automation in analyses. An alternative approach is to have safety properties formally verified using model-checkers. However, in this approach the analysis is typically inductive or requires exhaustive state-space exploration, which makes the analysis prone to combinatorial explosion and reduces the scalability of the approach (Wolforth *et al.*, 2010a). At a higher level there is the safety case; a set of complex documents which put forward arguments and evidence of a systems' dependability, gathered from sources including the results of other analyses. Here there have been significant developments with regards to reusability and compositionality.

In summary, there are many approaches to checking that a system meets its safety requirements. The classical methods developed from various industries (chemical, nuclear) have yielded useful methods that are still used today. Recognising some of their drawbacks, improvements have been made to allow more manageable analyses of larger scale, more complex systems, typical of those found in modern day safety critical systems. The most important improvements seen with contemporary approaches are:

- enhanced failure notation
- improved representation of failures and architecture
- compositionality
- automation to reduce the manual effort and improve accuracy

Clearly, improving reusability and managing reuse effectively are important and current areas of research modern safety analysis techniques. To help better understand the issues of reuse, the next section examines successful techniques for reuse found in other, more mature domains.

2.3 Reuse

Reuse is a widely used and very useful concept in many areas of modern engineering. There are various methods, types and scope for reuse, but reuse can generally be defined as the repeated application of some aspect of a system within the same or across various designs. The major motivation for reuse is often financial; being able to reuse a solution can save large amounts of money on resources and implementation costs. As for the implementation itself, there is also the potential benefit of reusing the experience and knowledge of the operational factors such as reliability and maintainability of a particular component. This is a particularly important consideration when reusing mechanical components. For example, once the design of such a component had proven its robustness in an operating environment, identical components could be included in other systems with a degree of confidence. For safety critical applications, these factors are extremely important, so there is a motivation to reuse wherever possible.

In the ideal scenario, reuse would allow the transfer of information across a variety of applications, but in practice it is very difficult to achieve. For a mechanical device, the operating environment can have a major influence on the performance of the components. Knowledge gained from one environment may not necessarily apply to another, for example due to a difference in ambient temperature. To overcome such practical problems, and in recognition of the benefits of reusability, mechanical components are typically designed to operate within tolerances. Designing a component to operate within a range of tolerances, rather than to fit a specific value, improves the likelihood that the component can be reused in other environments. This is an example where a *generalisation* of a property can increase the scope for reuse. This concept of generalisation is not limited to mechanical engineering; electronic and even software components can be constructed with a more general specification meaning they can be more easily reused.

In the remainder of this section, the scope of reuse within safety critical systems and safety analysis is summarised, and the level of reuse achievable in more mature domains is discussed, with a particular focus on software engineering.

2.3.2 Reuse in safety critical systems and analyses

Though careful design is essential, reuse (often *ad hoc*) still occurs within safety critical systems (Bush and Finkelstein, 2001). The reason is that the general advantages of reuse in other fields still apply, such as time and cost savings, but perhaps more importantly, the greater benefit of reuse within dependable systems is the 'trusted' aspect. If a solution is known to be dependable, perhaps from an observation of its operation as part of another system over a period of time, then it is often justifiable to apply the solution to the same problem when it is encountered again. More generally, it is desirable to use components that have well-known safety properties.

The assumption for subsequent safety analyses is that if components are reused, any associated information about safety properties can also be reused. However, because a change in the system context may introduce new factors, for example further influence from other components, it is not safe to assume that safety properties would hold. So even where component reuse is possible, failure behaviour will likely need to be reassessed in new applications. Re-specifying behaviour is no trivial task; despite attempts at generalising the descriptions of component failure behaviour, in most cases the methods still require context-specific descriptions. The FPTC method for example has improved on this by including some abstract syntax elements; generic references can be made using special symbols in expressions. Furthermore the compositional approach of modern safety analysis methods includes reusable elements, but the actual level of reuse achievable is fairly limited.

Levels of reuse can be put into classifications (Karunanithi and Bieman, 1993). If an element is reused without any modification, it is known as verbatim reuse. In other cases, reuse can take place subject to some modification, which is known as leveraged reuse. Trivial reuse is a variation of leveraged reuse, restricted to cases where only small modifications take place (Smith and Harrison 2005). Recognising this, allowing some flexibility with regards to what is reused makes at least some degree of reuse more feasible, but the difficulty then lies in how to carry out reuse appropriately.

In (Kelly and McDermid, 1997), the problems of manual safety case construction are discussed. They highlight the *ad hoc* style reuse that is commonly found; an analyst decides if a problem is ‘similar enough’ to another that a prior solution can be used in a new application. Though this may occasionally be successful, it ultimately becomes a ‘cut & paste’ approach to reuse and is simply unreliable, potentially leading to bad implementations. The idea that this kind of reuse is common in other types of analysis is suggested in (Bush and Finkelstein, 2001). It can be inferred from this work that a lack of systematic management is a major obstacle towards improving the scope of reuse in safety analysis.

Extensions to safety cases (Kelly and McDermid, 1998) have introduced a pattern concept to enable a degree of reuse by structuring how a safety case could be developed using prior arguments. It works by giving a documented solution to a common construction problem as a set of systematic guidelines for the analyst to follow. Although safety cases operate at a higher level, this approach to reuse can also apply to compositional safety analysis techniques.

There can still be some concerns even with a systematic approach to reuse however; supporting experience must be gathered first, and correctness must be rigorously checked. This still leaves the questions: where does responsibility lie if a problem is discovered? Are problems a result of a design fault or an inappropriately reused component? In the scope of this thesis, these concerns cannot be resolved; a certain degree of responsibility is assumed on the part of the analyst to ensure correctness.

2.3.3 Reuse in Software Engineering

Contemporary safety analysis introduces compositionality, along with partially automated techniques to improve the reusability and reliability of analyses, but their overall success has been limited. The focus of this thesis is to develop a language for describing failure behaviour that allows greater reuse of behaviour expressions than currently found in contemporary methods, along with a robust mechanism for the well-managed application of reuse. Based on this, as the investigation leads to generalised and abstract language constructs, it is useful to look at developments in the field of software engineering, which is a more mature domain and includes extensive mechanisms for managed reuse of program code.

Reusability is a very important and extensively used concept within software development. Modern software projects can be very complex and require programs with millions of lines of code. Creating programs of this size has forced developers to adopt more rigorous approaches to software design.

Improving the scope of code reuse was the primary aim of developing object-oriented (OO) programming. Bringing functions and associated data together created the concept of classes, which is the core of the OO paradigm. The idea is that a class becomes a reusable element, possibly a trusted solution to a software problem. To an extent, OO design can bring compositionality to software. It also provides the foundation for the more advanced reuse features. For example, a powerful technique for managed reuse within OO programming is the concept of *inheritance*. Inheritance allows new classes to be created from existing classes by specifying a hierarchy of relationships. When one class inherits from another, it means that certain properties are shared between them. A new class (the derived class) can inherit data and/or functions from an existing class (the base class) thereby reusing existing specifications. A class hierarchy can be constructed in several ways, giving rise to different types of inheritance. The simplest form is single inheritance, where there is a single base class inherited by a single derived class. This simple hierarchy can be extended further with the derived class itself being inherited to create another class, known as multi-level inheritance. A more complex hierarchy can go further and create classes which take attributes from more than one type, called multiple inheritance, where a derived class inherits from more than one base class. When these types of inheritance, multi-level and multiple, are used within the same hierarchy, it is known as hybrid inheritance. As well as for code reuse, inheritance is a useful method for managing changes, for example any change made to the parent base class will be immediately inherited by all derived classes.

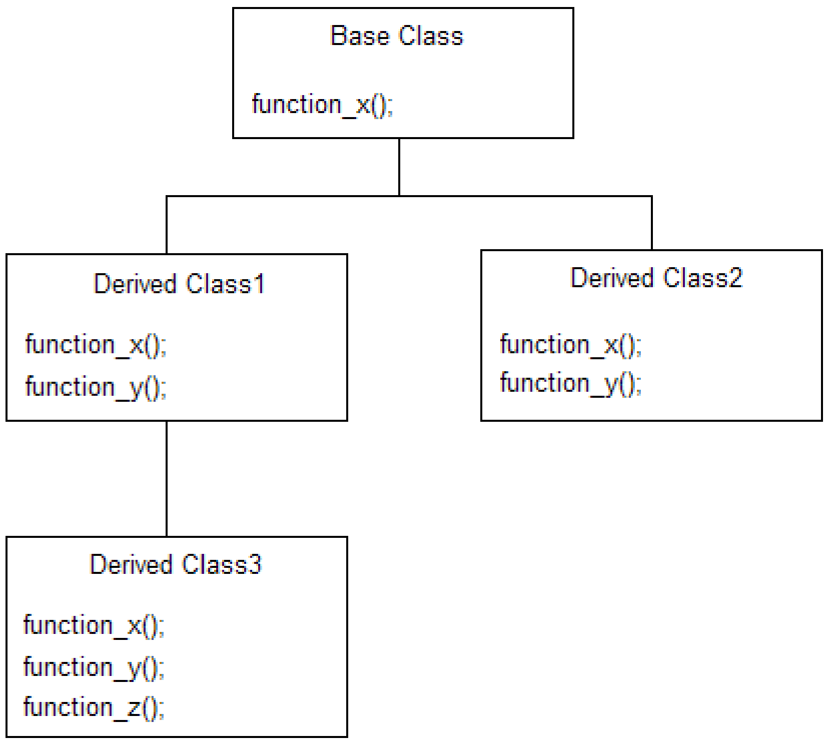


Figure 10 Inheritance in object-oriented programming

For example in Figure 10, all derived classes contain function_x(). This is a useful feature for creating specialisations of a type which are all compatible within the inheritance hierarchy.

At a higher level of software design, there is another type of reusable element, the *design pattern* (Gamma *et al.*, 1995). These are structured combinations of interacting classes that when implemented, are known to provide solutions to recurring design problems. A design pattern consists of a diagram, illustrating the flow of control and data between classes, and documentation about how and when the pattern should be applied. Design patterns capture solutions that have evolved over time, based on acquired expertise, experience and successful approaches to prior software problems. By providing this knowledge in a structured way, with appropriate documentation to guide their usage, design patterns can speed up software development.

Choosing an applicable pattern for a solution is itself no easy task, so as a collection they are catalogued and organised into categories that define the purpose and scope of a pattern. Creating a catalogue like this makes it easier to find the solution to a problem, giving a good foundation to managing appropriate reuse.

There are some drawbacks with using design patterns, the most obvious being that they are not directly implementable solutions; they serve as *knowledge* reuse rather than *literal* reuse and are therefore a template for developers to base their implementation on (Gamma *et al.*, 1995). The idea is to show the interactions of classes for commonly occurring problems, but it is left up to the developer to follow the implementation guidelines and provide the actual program code. Potentially this undermines the reusability, at least in terms of a reliable solution, as it can easily introduce errors with the manual implementation.

2.4 Summary of literature review

There are several methods that are used for safety analysis. The classical methods have evolved over time, leading to the state-of-the-art contemporary methods available today that are capable of modelling complex, compositional systems. Some are based on failure logic models, using deductive methods to trace failures through a system. Alternatively there are methods which perform formal verification of models to inductively check safety properties.

Performing a safety analysis of these types of system typically requires a team of analysts. It is often expensive and time-consuming, yet it is still a necessary process. Some methods allow a degree of automation, to aid analyses and reduce the possibility for human error, but there is still a manual element to each technique. This is partly the motivation for attempts to improve the way safety analyses are performed, the most prominent being the model-based approach.

Reusability has been a major influence in the development of these methods. To reflect the compositional architecture of designs, analysis techniques have themselves become compositional in response. The aim is to reflect the same type of literal reuse of architectural components as similarly reusable analysis elements, for example reusing associated definitions of 'well-known' component failure behaviour across analyses without modification. This is particularly desirable in the domain of safety critical systems where there is the motivation to use 'trusted' components within designs. However, achieving this type of reuse has proven very difficult with a suitable degree of confidence within current methods. State-of-the-art safety analysis methods, rather than focussing on literal component reuse, make some attempts to capture and reuse knowledge of component failure behaviour, often defined as a logical expression.

One aspect where problems arise is with context-specificity of definitions of failure behaviour which prevents information from one application being used in another design. Generalisation and abstraction seen in some methods are concepts used to improve reusability by attempting to remove the context dependence. The intention is that a more general analysis element would be applicable across a range of studies. However attempts at abstraction or generalisation in any scenario require additional documentation to guide the user in its application, thus it is also essential to consider how reuse is actually performed.

The limiting factor here is an overall lack of a well-defined, managed approach to reusing safety analyses. The prime example is the 'cut & paste' approach to reuse where the analyst makes the (possibly incorrect) observation that certain components are similar enough for failure data to be reused when in fact it is not appropriate to do so. The lack of rigour in the process means that when reuse is performed, it is often in an *ad hoc* manner. These are the two most important problems that must be considered in a new approach; enabling knowledge reuse of component failure behaviour and establishing a systematic method for performing reuse.

The second part of the chapter focused on reuse in general, with attention paid to reusability concepts as found in software engineering. The concept of inheritance was of particular interest in this study, being a mechanism that can be used in order to manage reuse more effectively. Inheritance enables reuse by creating a specialisation derived from a more general base type. Employing this within safety analysis and component annotations, creating a reuse mechanism based on inheritance may provide an effective way to manage the reuse process. Another useful feature found in software engineering is Design Patterns; reusable elements that capture general solutions to commonly occurring software problems, and provide documentation along with a diagram to guide their implementation. They are not directly implementable themselves, rather they are an example of where knowledge has been captured and expressed in a reusable format.

Attempting to enable these kinds of features within a safety analysis method necessitates that a more abstract approach to expressing failure behaviour is developed. Lessons learned from the literature review imply this can be done via the creation of a generalised language for the description of component failure behaviour within an established safety analysis method. Inductive methods (i.e. formal verification) were deemed unsuitable for this development, mainly due to their inherent problems of combinatorial explosion.

The HiP-HOPS technique is a deductive method and a good candidate for this development because its annotation syntax can be easily extended and provide a more easily readable syntax. A generalised syntax will allow collective, generic statements to be made about failure behaviour which should hold in any application context, but it is important to realise this may not be possible for all components and failure types. As a final step to ensure appropriate and managed reuse, it is essential to develop a system of documentation to guide the application of generalised expressions. Taking all these points into account, the following requirements for an extended language are as follows:

1. The syntax must allow *generalisations* to be made which can define behaviour, as far as possible, independently of context
2. Can be *automatically* parsed without sacrificing readability
3. Be *well documented*, including systematic rules for guiding reuse
4. Remain compatible with the current annotation language within HiP-HOPS

In conclusion, the aims of the thesis have been focused and a set of requirements for what a new approach should include have been defined. In the next chapter, these requirements are taken and developed into a new method for managed reuse alongside a robust safety analysis technique; a reusable, generalised language extension to the HiP-HOPS tool.

3 A Generalised Failure Logic extension to HiP-HOPS

The first part of this chapter explores the HiP-HOPS analysis process to discover the limitations of the classical component annotation syntax. Following this, it is shown how generalisations of component failure behaviour can be made, captured and expressed within an extended annotation syntax. Further, it is shown how sets of failure behaviour expressions created with the generalised syntax can be encapsulated and managed with documentation so that a pattern concept can be applied to component annotation.

3.1 Safety Analysis in HiP-HOPS

Performing a HiP-HOPS study on a system involves three main phases:

- System modelling and failure annotation
- Model parsing and fault tree synthesis
- Fault tree analysis and FMEA generation

The first stage is manual and comprises development of the system model and the establishment of the local failure behaviour for each component. The local behaviour describes how the component responds to failures generated internally or received at its inputs having been propagated from other components. In HIP-HOPS, this is expressed as effects of internal and input failures on the component's own outputs. A variant of HAZOP (Kletz, 1997) is used to initially identify the possible component output failures and their causes. During this preliminary study, the types of failures, the set of *failure classes*, to be examined can be defined. These types will depend upon the context and application of the system, but in the general case these can be the familiar HAZOP-style classifiers, i.e. provision, value, timing.

In the classical annotation syntax, there is the general assumption that every component has a set of inputs and delivers a set of outputs. This forms the basis of the logical expressions which relate deviations of output to their respective causes and allows the set of fault trees for the system to be synthesised. Component failure behaviour is annotated into a model as a set of logical expressions; these can include Boolean operators such as conjunction (AND) and disjunction (OR) to describe combinations of failures, for example:

```
Omission-o1 = Omission-i1 OR Internal_Failure
```

An expression like the above relates an output deviation (the left-hand side) to its causes (right-hand side), which can be deviations of inputs, internal malfunctions, or any combination of these. In this case, the output deviation, the omission of output `o1`, is caused by an omission of input `i1` or by a basic event, an internal failure of the component. More detail can be included by specifying the parameters of the deviations; connections between components may have more than one attribute, which necessitates the definition of the extra detail. For example in a hydraulic system, the fluid flow, pressure and temperature may all need to be monitored between connected components, therefore expressions may be needed which refer specifically to an individual attribute, e.g. a low value for the pressure may cause a failure of the hydraulics to operate correctly.

Once the modelling and subsequent annotation is complete, HiP-HOPS performs its automated analysis steps. The topology of the model is used to automatically determine how the local failure behaviour, specified by the component annotations, causes failures to propagate through connections in the model and ultimately cause functional failures at the system output. This global view of failure is captured deductively (effects to causes) in a set of fault trees, synthesised from the model traversal. Finally, qualitative or quantitative analysis can be automatically performed on the fault trees to establish whether the system meets its safety or reliability requirements. A table equivalent to a multiple failure FMEA can then be automatically obtained from the fault tree logic, recording for each component in the system and for each failure mode of that component, any direct effects on the system and any further effects caused in conjunction with other failure events.

As HiP-HOPS is capable of providing safety information at each stage in the evolution of a design, it would be useful to reuse expressions of failure behaviour, where possible, to save the effort of re-annotating components in all design iterations. The more general a failure expression is, the more easily it could be reused in this way. The major difficulty with this, and with deviation-based failure logic modelling in general, is that the modelling is dependent on the nominal behaviour of the system and the nominal behaviour is then dependent on the application context and state. In practice this means that the nominal behaviour can change and so reliable reuse of specific failure logic is difficult to achieve. However, creating generalisations of failure logic can be achieved and thus help guide and improve the efficiency of creating models in step with designs.

3.1.1 Fuel System example

An example of HiP-HOPS analysis of a simple fuel system is given in this section to highlight what is currently achievable with the annotation syntax, its limitations, areas where the syntax could be extended to enable a greater degree of generalisation and therefore where possibilities lie to achieve greater reuse of failure expressions.

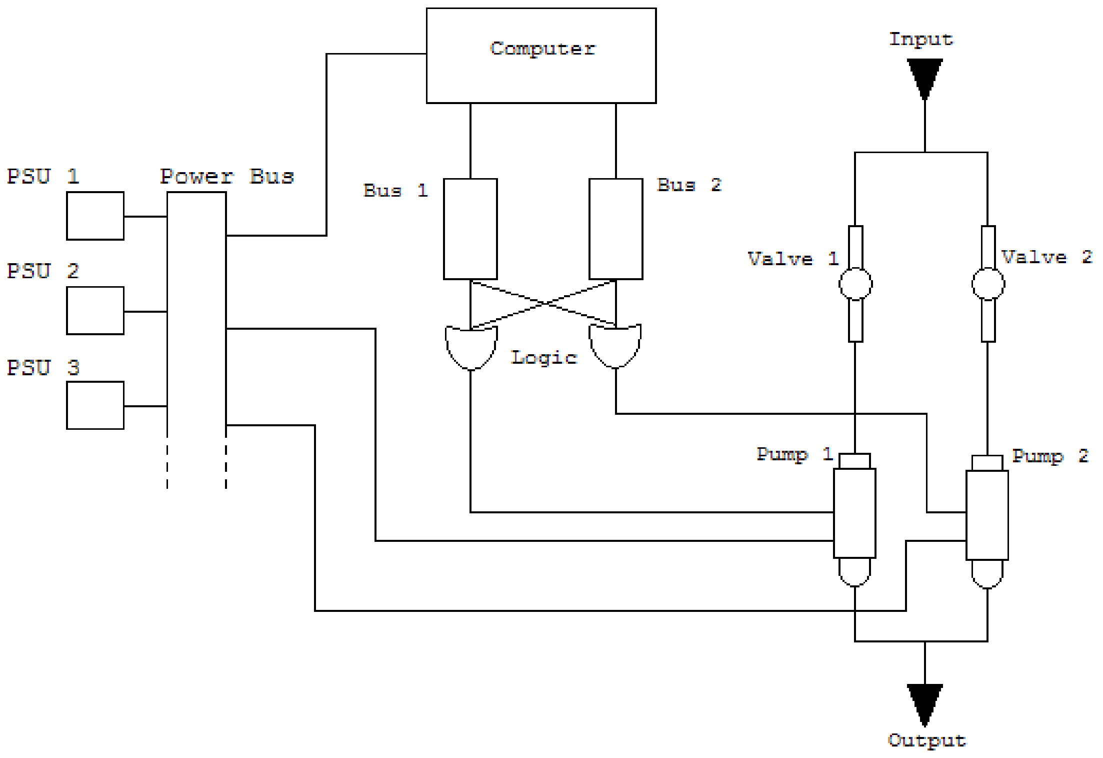


Figure 11 Fuel System

The fuel system in Figure 11 is designed to continually provide an output flow of fuel, with a provision for some fault tolerance. Pathways for power, control and fuel flow are replicated for redundancy, so that fuel will continue to flow as long as there is sufficient power to the system and a correct control signal is delivered to an operational pump. Control signals pass through an embedded logic system that can compensate for a permanent failure of one bus. Any missing control signal from one bus is synthesised from the other bus, meaning signal omissions are only propagated when both buses fail together. Commission failures (an unintended signal) cannot be detected or corrected with this logic system, so these will always propagate to the pumps. The use of two pumps means that an omission of output can only occur if both pumps fail, whereas a commission of output can occur if only one pump experiences a commission failure.

PSU
`Omission-out = PSU_failed`
Power Bus
`Omission-out1 = Omission-in1 AND Omission-in2 AND Omission-in3`
`Omission-out2 = Omission-in1 AND Omission-in2 AND Omission-in3`
`Omission-out3 = Omission-in1 AND Omission-in2 AND Omission-in3`
Computer
`Omission-out1-sig = Omission-power OR computer_failure`
`Omission-out2-sig = Omission-power OR computer_failure`
`Commission-out1-sig = memory_stuck`
`Commission-out2-sig = memory_stuck`
Data Bus
`Omission-out-sig = Omission-in-sig OR bus_failed`
`Commission-out-sig = Commission-in-sig`
Logic
`Omission-out-sig = Omission-in1-sig AND Omission-in2-sig`
`Commission-out-sig = Commission-in1-sig OR Commission-in2-sig`
Valve
`Omission-out = Omission-in OR stuck_closed`
`Commission-out = Commission-in OR stuck_open`
Pump
`Omission-out = Omission-in OR Omission-control OR`
`               Omission-power OR stuck_closed`
`Commission-out = Commission-control OR stuck_open`

Table 3 Fuel System annotations

The system as a whole is powered by a series of redundant power supply units (PSUs) routed through a single power bus. This design allows for additional redundant PSUs to be added as required, or for failed PSUs to be replaced without power disruption to the system. This feature also allows provision for a finalised number of PSUs to be specified in a later iteration of the design. Individual PSU failure can be tolerated, but multiple PSU failures will result in total power loss for the system.

This case study focuses only on two classes of failure, omission and commission; therefore the system failure modes studied are the omission or commission of fuel flow. For example the manner in which failures propagate through the system means that PSU failure can lead to pump failure and hence an omission of output. The full failure behaviour for the system is given in Table 3.

The PSU has a single failure mode with one application-independent cause; an internal failure stops the PSU from providing power at its output, which could occur within any system which uses the PSU. It is possible that as each PSU in the system is identical, they can share the same annotation, which is true in the context of this example. Likewise for other replicated components within the same system, they can share the annotations of their identical counterparts. Those expressions can be stored and re-applied whenever the associated component is used in a model.

The power bus has its failure behaviour described, as in Table 3, with three expressions. Observing the common structure within each of these expressions, there is a common behaviour; every output deviation is brought about by the same input deviation cause. Using natural language, it is possible to describe the component's behaviour in a more general manner '*an omission of any output is caused by an omission of all inputs*'. This generic behaviour has no dependency on the application context and will therefore hold in any system that incorporates the power bus. Therefore it can be considered to be a very simple, reusable pattern of behaviour. Indeed it is possible to see more forms of general behaviour and patterns in the example system.

The computer can experience omission and commission failures on both its outputs, and each type has the same cause on either port. For instance the commission failure on either output port is caused by the same internal 'memory stuck' failure. In more general terms, it could be said that the ‘memory stuck’ failure causes a commission on *any* output port.

The individual data buses in this system will propagate any type of failure received at its input to its output. In this case the activation signal ‘sig’ for the fuel pumps can either be omitted or present though undesired; this is equally true for any other signal that the data bus component may transmit in any context, thus it can be generally said that the *same* erroneous input signal will be propagated to the output. Furthermore, for this example component, the behaviour will hold for any number of inputs and outputs the bus may have, so it can be said that an omission on *any* input of a signal will propagate as an omission of the *same* signal at a corresponding output, and likewise for commission failures.

A logic controller in this system will only propagate an omission failure if it occurs on both its inputs. For instance an omission of output is caused when an omission occurs on *all* inputs, a conjunction of signal omissions on ‘in1’ and ‘in2’. A commission failure will propagate when at least one input, i.e. *any* input, receives an unintended signal.

The valves controlling the input flow of fuel have a very basic failure behaviour which is valid for any system in which one is used. For instance the valve can experience basic failures of being either stuck open, leading to a potential commission of output flow, or stuck closed, leading to a potential omission of output flow. The precise material whose flow is being controlled is not relevant for an expression of the valve’s failure behaviour, thus the existing annotations will hold in any application and do not need to be further generalised; they can be stored and reused along with the valve component.

The pumps will fail to deliver an output in response to an omission of any of its inputs, the input flow, the control signal or the power source, or if the pump itself is stuck closed. Undesired output from the pumps can occur if there is an undesired activation of the pump control or if the pump is stuck open.

Under analysis in HiP-HOPS, potential design flaws in the system would be discovered, such as the single point of failure ‘computer failure’. This may force design changes to be considered, leading for example to the addition of a redundant backup computer to provide a fail-safe mechanism in the event one computer suffers this failure, avoiding an otherwise complete system failure.

3.1.2 Limits of HiP-HOPS component annotation

The general case is that component annotations as in Table 3 have very limited applicability, e.g. the expressions only apply to this system and are not transferrable into another study, but there are situations, like with the PSU and valve components, where it is possible to reuse annotations developed with the current syntax. Even so, such reuse is somewhat *ad hoc* and clearly not subject to a rigorous process; an aspect identified previously as being essential to achieving a suitable method for reuse.

Capturing behaviour using the original HiP-HOPS annotation syntax typically requires multiple expressions containing combinations of events. This is sufficient to fully describe a system’s failure behaviour, as seen with the fuel system example; however it can become a burden, particularly with regard to reusability of expressions. For instance the power and data buses have a failure behaviour which is independent of the application context, however their annotations would have to be completely reconstructed if additional components were connected to them in a modified design to take into account the extra potential failure causes and effects. This highlights the limit of reusability within HiP-HOPS. The effort of reconstructing component annotations for every application should not be underestimated (this can easily range into the thousands of expressions), so being able to have more general annotations, thus improving the scope of reuse, is very desirable.

Most of the components in this example system can have their failure behaviour described in a general way using statements such as ‘any output’ or ‘all inputs’. Beyond this small example, many electromechanical and complex programmable components exhibit similar patterns of behaviour that would be useful to capture. The Time-Triggered Protocol (TTP) communication controller (Kopetz and Grünsteidl, 1994) for instance is designed to always fail silent in response to detectable omission, commission and timing failures in received messages, and this behaviour is independent of the number, type or relative scheduling of these messages. Fail silent behaviour as in the TTP controller is a transformation of one failure type into an omission of output i.e. the failure type is not propagated. Generally speaking, the fail silent behaviour can be described as the transformation of any input failure class into an omission of output.

Though it is possible to describe behaviour generally with natural language, the original annotation syntax of HiP-HOPS clearly does not have the ability to make such generalisations within failure behaviour expressions. In order to capture such patterns of behaviour, the syntax must be extended with more abstract terms. The potential to make generalisations within component annotations lies in making collective references, such as ‘any’ or ‘all’, to the terms in a given expression. Furthermore, an abstract definition of failure behaviour enables a far greater potential for reuse of component annotations, for instance in the scalability of expressions; a reference to ‘all inputs’ has no dependency on the specific number of input ports and thus is unaffected by design changes that alter this quantity.

3.2 The Generalised Failure Language

This section proposes the extensions to the HiP-HOPS syntax for component annotation, known as the Generalised Failure Language (GFL), which allows the creation of Generalised Failure Expressions (GFEs). GFEs make it possible to capture patterns of behaviour which can be automatically interpreted within a compositional safety analysis process.

The syntax for the GFL is fully defined in section 3.2.1. Any annotation which employs the GFL must contain a component identifier, any inherited values and one or more generalised expressions. A generalised expression in this sense does not necessarily contain abstract generalised terms, as a classical HiP-HOPS expression may be sufficient in some cases. Alternatively it could be a more complex expression containing both classical and generalised terms, which is allowable in the syntax.

3.2.1 Grammar of the GFL

Below is the complete grammar for the new syntax, giving the rules on how to construct valid expressions in GFL format. Detail of the formal semantics of GFL can be found in (Wolforth *et al*., 2010a).

```
ComponentAnnotation =    ComponentName [InheritDirective] GFE {GFE}

ComponentName       =    "NAME" ComponentID

InheritDirective    =    "INHERIT" ComponentID

ComponentID         =    Identifier

GFE                 =    GOD "=" InputExpression

GOD                 =    OutputFailureClass "-" OutputPort ["-" OutputParam]

InputExpression     =    OrTerm { "OR" OrTerm }

OrTerm              =    AndTerm { "AND" AndTerm }

AndTerm             =    ["NOT"] (GID | "(" InputExpression ")" | BasicEvent)

GID                 =    InputFailureClass "-" InputPort ["-" InputParam]

BasicEvent          =    Identifier

OutputFailureClass  =    "FC" [Exception] | FCList | FailureClass

InputFailureClass   =    Operator "(" ( ("FC" [Exception]) | FCList ) ")"

                    |    "SAME(FC)" | FailureClass

FCList              =    "FC:" List

FailureClass        =    Identifier

OutputPort          =    "OP" [Exception] | OutPortList | PortName

InputPort           =    Operator "(" (("IP" [Exception]) | InPortList )")"

                    |    PortName

OutPortList         =    "OP:" List

InPortList          =    "IP:" List

PortName            =    Identifier
```

```
OutputParam         =    "PM" [Exception] | ParamList | ParamName

InputParam          =    Operator "(" ParamList ")"

                    |    "SAME(PM)"|ParamName

ParamList           =    "PM:" List

ParamName           =    Identifier

Exception           =    "EXCEPT" List

List                =    "{" Identifier { "," Identifier } "}"

Operator            =    "ALL" | "ANY" | "MAJ"
```

Where "Identifier" is a letter or underscore followed by zero or more letters, numbers, or underscores, that does not contain a case sensitive match of any of the reserved Keywords (`NAME, INHERIT, OR, AND, NOT, FC, OP, IP, EXCEPT, ALL, ANY, MAJ`) or their long versions. Terminals are presented in double quotes (""). For convenience or clarification, some terminals can be replaced as follows:

```
FC  = FAILURE CLASS

OP  = OUTPUT PORT

IP  = INPUT PORT

PM  = PARAMETER

NOT = !

OR  = +

AND = *
```

3.2.2 Sets of values

A GFE, like a classical HiP-HOPS expression, consists of an output deviation (the left-hand side) and its causes (the right-hand side), typically a combination of input deviations and/or basic events. Either side may be generalised by making collective references to failure class, port and parameter terms; these terms retain their meaning from classical HiP-HOPS as explained in section 3.1. Making these terms abstract is the core of the generalised syntax. However, an expression does not need to contain solely abstract terms in order to be general, or to be compatible with this extended syntax. If for example there is only one class of failure a component can experience, then it is sufficient to make this reference directly in the expression.

The first term, the failure class, refers to the type of failure. Within HiP-HOPS, the generalisation is made that every component operates on a set of inputs and delivers a set of outputs, which means there are a limited number of failure types a component can experience. These fall under the category of Provision (e.g. Omission, Commission), Timing (e.g. Early, Late) and Value (e.g. High, Low). This work uses these generic categories for failure classes for explanation and demonstration; in practice these can be more specific if required. In the interests of improving reusability, the use of general categories and values should be encouraged where possible.

Thus the language proposes the use of the abstract term to represent failure classes in an expression. This identifier will represent the set of valid failure classes the component can experience in any particular application. For example, the output deviation:

```
Omission-out-sig
```

could become:

```
FC-out-sig
```

This abstract reference ‘`FC`’ (for failure classes) will be replaced with a specific value when analysed, yielding the necessary application-specific expression; recursive instantiation with all set values will create a set of expressions. At this point the assumption is made that these values can be retrieved from the model when expressions are parsed during the automated analysis. The parser will instantiate expressions using all values that a set contains therefore sets do not need any specific ordering.

Next there is an abstraction made for ports, working on the same basis as for failure classes. Input and output ports are represented with ‘`IP`’ and ‘`OP`’ respectively. Finally there is the optional parameter, which is made abstract with the ‘`PM`’ term. Below are some examples of how these terms would be applied:

```
Omission-o1 = Omission-i1
Omission-o2 = Omission-i1
Omission-o3 = Omission-i1
```

Using sets, these expressions can be generalised as:

```
FC-OP = Omission-i1
```

where `FC:{Omission}` and `OP:{o1, o2, o3}`.

Alternatively, the syntax allows values contained in a set to be manually specified within an expression:

```
FC:{Omission}-OP:{o1,o2,o3} = Omission-i1
```

The `FC` set as defined in the above expression contains only a single value. This is allowable in the syntax but not necessary as an abstraction; a specific value can be used directly as on the right-hand side (’`Omission`’ and ‘`i1`’). Instantiation with all possible values as referenced by each set would yield the original three expressions above[3].

The benefit of this notation becomes clear when for instance the data bus component as in Figure 11 were to be modified to have a hundred inputs and a hundred outputs. With the current syntax, it would require a hundred expressions. With the generalised syntax, the improved scalability of an abstract reference ‘`OP`’ means only a single expression would be needed, with information about ports being automatically parsed from the model.

3.2.2.1 EXCEPT

By default, every value that a set contains is used to create expressions, but this process can be made more flexible by restricting the scope to a limited set of values in certain situations. For example, it may be more useful to define those values which *do not* apply, rather than listing all that do apply, so a special modifier is introduced to allow *exceptions* to be made.

Consider a hypothetical bus component which has two inputs and ten outputs. An omission of the first input causes an omission of the first nine outputs, whereas an omission of the second input causes an omission only of the tenth output. Even using abstract terms, describing this behaviour would need ten separate expressions. Using sets with the additional EXCEPT modifier, the behaviour can be described with just two expressions:

```
Omission-OP EXCEPT:{out10} = Omission-in1
Omission-out10 = Omission-in2
```

[3] The process of instantiation is described in detail in section 3.2.5.

The set `OP` still represents all output port values, but the value `out10` is excluded during instantiation of the first expression.

3.2.3 Operations on sets

Combinations of failure causes means that the size (i.e. the length) of expressions can grow very large, for example a component may have a hundred inputs, and an output failure caused by a combination of failures of all inputs. This aspect can be a hindrance to reuse, as longer expressions become more difficult to maintain. The next development in the GFL is focused on making generalisations where output deviations have more than one cause, for example a combination of causes as in the fuel system’s power bus behaviour:

```
Omission-out1 = Omission-in1 AND Omission-in2 AND Omission-in3
```

Previously stated was the desire to capture general statements of behaviour with terms such as ‘any output’ or ‘all inputs’, which in the hypothetical case of a component with a hundred inputs could be very beneficial. Using sets, it is possible to reduce the number of individual expressions needed for specification. By manipulating sets with certain operators, the language becomes capable of *contracting* expressions to reduce their length. Contracting expressions is not a generalisation in order improve reusability *per se*, however it does make a useful contribution to the extended syntax by enabling a degree of scalability within GFEs and makes larger expressions more manageable.

The accuracy of analyses is essential, so contraction is only ever performed where the logical meaning of an expression would not be altered. Therefore the proposed operators perform contraction as a type of ‘factorisation’. No essential information is removed from the expression; the structure of the expression merely changes to a more compact form, while maintaining all values and logical combinations. Contractions are made by grouping together common terms combined under the same logical operation, for example:

```
Omission-in1 AND Omission-in2 AND Omission-in3
```

can be ‘factorised’ to create a contracted form:

```
Omission-(in1 AND in2 AND in3)
```

Further contraction can be made here if a set term is used to represent the list of input ports (‘`IP`’) in conjunction with a specification of the common logical operator:

```
Omission-AND(IP)
```

Clearly, where an output deviation has many combined causes, using operators can significantly reduce the size of resulting expressions.

Applying an operator to a set means they are given the same scalability as abstract sets while allowing the capture of general behaviour. There is also the option to apply the EXCEPT modifier here to limit the scope of an operator. The use of operators therefore conforms to the general format:

```
operator(set except:{list})
```

Logical combinations only exist in an expression on the right-hand side. For this reason, operators only apply to input deviation terms and are never applied to sets in an output deviation. Moreover, any set term used in an input deviation *must* have an operator applied to it, because an abstract set in an input deviation always implies either a logical combination of values or a correspondence between input and output values (a special case which requires its own operator). When only a single specific value is needed however, a set and operator are not required.

Keeping to the overall aim of improving and encouraging reuse, keywords are used to define operators rather than abstract symbols. With carefully chosen and defined operators, the extended syntax is more easily readable and closer to a natural language. Many operators could be introduced into this syntax, but for this work the GFL has been developed with four operators deemed the most useful for investigation. Two of these are operators which generalise Boolean AND and OR logical combinations. Also included is an equivalence operator used to define a relation i.e. propagation of values across a component's input/output. Finally there is the majority voter operator to show how more complex logical behaviour, beyond that seen in the previous example system, can also be generalised in an expression.

3.2.3.1 ANY

An individual pump in the fuel system (Figure 11) continually receives three inputs; power, control and fuel and provides a single output, the fuel flow. The pump is designed such that an omission of any input or a 'stuck closed' internal failure will result in an omission of its output:

```
Omission-out = Omission-in OR Omission-control OR
               Omission-power OR stuck_closed
```

It is possible to summarise this behaviour as '*omission of output is caused by an omission of any input or by an internal failure*'. Here there is an abstract logical disjunction of input ports, described generally as '*any input*'.

The first operator added to the language is ANY, to represent a logical disjunction of a set's values. This is equivalent to combining values with OR logic, but this operator is applied to 'loop' through all the possible values to automatically create combinations.

So applying the ANY operator to contract the above expression, it can be rewritten in the generalised form:

```
Omission-out = Omission-ANY(IP) OR stuck_closed
```

where the set `IP` contains the three input ports (in, control, power).

3.2.3.2 ALL

Returning to the power bus component as an example, the component experiences an omission failure in response to a combination of causes:

```
Omission-out1 = Omission-in1 AND Omission-in2 AND Omission-in3
```

Looking at this behaviour, it can be said that '*omission of output is caused by an omission of all inputs*'. Here there is an abstract logical conjunction of input ports, described generally as '*all inputs*'.

The next operator in the language is ALL, which represents a logical conjunction of a set's values. It is equivalent to a series of values combined with AND logic. The power bus behaviour can now be rewritten in a generalised form:

```
Omission-out1 = Omission-ALL(IP)
```

where the set `IP` contains the three input ports (in1, in2, in3). In this example, if additional PSUs were attached to the power bus, the above generalised expression would remain valid. In most cases, ALL should not be applied to the `FC` set, as this may cause problems in the analysis to have a deviation which has more than one classification. For example, a semantic error can occur if ALL(`FC`) were used to form a conjunction of mutually exclusive failure classes, for instance omission and value; it is not possible to have no value and a value out of range on the same port simultaneously. Therefore the careful use of this operator is left to the discretion of the analyst.

3.2.3.3 SAME

So far the operators have been introduced to contract the structure of expressions, making them more compact and at the same time more reusable. In the examples provided above, there is often an equivalence of values on either side of an expression; for instance the failure class 'Omission' is shown propagating from input to output. This is not uncommon in typical component behaviour, for example where an omission failure propagates through a system. Consider a simple component whose behaviour can be expressed as follows:

```
Omission-out = Omission-in
    High-out = High-in
     Low-out = Low-in
```

This behaviour can be described generally as '*any class of output failure is caused by the same class of failure at input*'. This type of propagation is captured in the syntax with the SAME operator. Unlike the previous operators, SAME reduces the overall number of expressions, rather than reducing the size of an individual expression, but is still very useful with regards reuse and scalability. Thus with the SAME operator, the component's behaviour as above can be rewritten:

```
FC-out = SAME(FC)-in
```

where the set `FC` contains the three failure classes (omission, high, low). The SAME operator describes a correspondence, therefore the set to which it is applied must also appear as a term in the output deviation, seen in this example with failure classes (`FC`). It is subject to the same instantiation rules as determined in section 3.2.5, thus with instantiation the SAME term is replaced with a specific corresponding value e.g. failure class. Only one value is used; the set is not 'looped' as with other operators.

Additionally, SAME can only apply to failure class and parameter sets, as it cannot be assumed that there is a correspondence between ports; components do not always have the same number of inputs as outputs, nor do they necessarily provide the same function.

3.2.3.4 MAJ

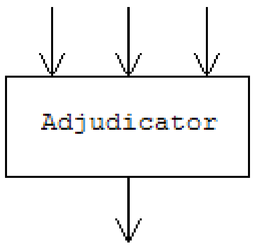


Figure 12 Example voting component

As a demonstration of more complex behaviour that can also be generalised, Figure 12 shows an Adjudicator which monitors three separate inputs and delivers an output based on a majority combination of input values, i.e. a signal must be present at two or more inputs in order to be propagated to the single output. The failure behaviour of the Adjudicator can be expressed as follows:

```
Omission-out   = Omission-in1 AND Omission-in2 OR
                 Omission-in1 AND Omission-in3 OR
                 Omission-in2 AND Omission-in3
```

The behaviour can be described generally as '*omission of output is caused by an omission of a majority of inputs*'. To represent majority voting behaviour, the MAJ operator is introduced. Thus the Adjudicator's behaviour can now be expressed as:

```
Omission-out = Omission-MAJ(IP)
```

where the set `IP` contains the three input ports (in1, in2, in3). The MAJ operator is a useful example of how more complex operations can be represented in the new language, as the original expression to describe the behaviour is considerably longer and more complicated than the generalised form.

3.2.4 Inheritance

The reuse mechanism developed for use with GFL is called *inheritance*, influenced by the similar concept found in OO programming (Cardelli and Wegner, 1985). Inheritance works on the assumption that the analyst will have a source library from which patterns can be inherited and reused. The mechanism allows patterns to be reused, either singly or extended by a process of specialisation, by using the `INHERIT` directive in an annotation.

The first type of reuse possible is where an existing set of expressions (a complete definition of a component's behaviour) is used with no changes to annotate a component. For example, the Direct Propagation pattern that defines propagation of failures is held in a library. A data bus component expresses this behaviour, and so brings this behaviour directly into its annotation:

<u>Direct Propagation</u>

```
FC-OP-PM = SAME(FC)-ANY(IP)-SAME(PM)
```

<u>Data Bus</u>

```
INHERIT Direct Propagation
```

The second type of reuse, reuse via specialisation, is a more complex, yet more flexible approach. The mechanism is similar to that found in OO inheritance. Specialisation uses a stored pattern to annotate a component that expresses an extended version or variation of the pre-defined behaviour. This works by taking the original or *base* behaviour, inheriting it into another component’s annotation, and then defining any additional behaviour to make the new annotation a specialisation of the original behaviour, for example:

Specialised Bus

```
INHERIT Direct Propagation
Value-OP-PM = Value-ANY(IP)-SAME(PM) OR InternalFailure
```

Using this type of inheritance creates complications when a specialised component uses some, but not all of the base type behaviour. This leads to another feature of the method, the ability to override behaviour. Inherited values are the ones which are overridden, i.e. they have the lower precedence. Any expressions explicitly defined in the specialised pattern will override any conflicting inherited values. It is equivalent to using the EXCEPT modifier, though this time it is applied automatically. For example, in the Specialised Bus above, the ‘value’ class of output deviations are overridden, making the pattern equivalent to:

```
FC EXCEPT:{Value}-OP-PM = SAME(FC)-ANY(IP)-SAME(PM)
Value-OP-PM = Value-ANY(IP)-SAME(PM) OR InternalFailure
```

3.2.5 Instantiation of GFL expressions

To accurately describe the way a component can fail, and therefore to provide the necessary detail for subsequent analysis, a component is required to have application-specific information available. In the case of HiP-HOPS, specific information is only required for fault tree synthesis, but is not essential at the annotation phase. Therefore, generalisation can be employed for component annotations, provided that context-specific information becomes available at later analysis stages.

In order to utilise a GFE, it must be *instantiated* by replacing abstract terms with application-specific values, automatically creating a set of failure expressions for a component under analysis. Any individual application-specific values may already be listed within a GFE, otherwise the values represented by sets can be automatically obtained from the model or defined by the analyst[4].

The instantiation of GFEs is automated and, in keeping with existing HiP-HOPS convention, works through expression terms from left to right; individual terms and operators are instantiated in the order `FC` then `OP/IP` then `PM`. The process begins with the output (left-hand) side; in a generalised form, these are known as Generalised Output Deviations (GODs). A GOD can represent multiple output deviations. All sets in the GOD terms are iterated through, creating a new output deviation for each value encountered. Where there are multiple sets, every possible combination of values must be instantiated. Consider the following example GOD:

```
FC-OP
```

given the sets:

`FC:{Omission, Commission}` and `OP:{out1, out2}`

the GOD will instantiate the following output deviations:

```
Omission-out1
Omission-out2
Commission-out1
Commission-out2
```

[4] In the latter case, *templates* are used to guide the documentation of such information (section 3.3.1).

Thus the GOD and associated set values force the instantiation of four separate expressions. Following this, the input (right-hand) side of the generalised expression, which is referred to as a Generalised Input Deviation (GID), must then be expanded. Whereas the GOD represents and creates multiple instances, the GID expands into a single expression, though it is not necessarily identical for each instance (when SAME is used). Any set used in a GID must therefore be subject to an operation so that it is expanded rather than iterated. As a result their expansion follows the rules as given in the previous discussion of the operators i.e. values in the set are combined according to the logical rules imposed by the operator. Consider the following example GID:

```
Omission-ALL(IP)
```

given the set:

```
IP:{in1, in2, in3}
```

will expand as the following input deviation:

```
Omission-in1 AND Omission-in2 AND Omission-in3
```

Any specific (non-generalised) terms are maintained throughout all instances. So a complete instantiation would proceed as shown in the following example steps:

Given the GFE for a bus, propagating omission failures from input to output:

```
Omission-OP-PM = Omission-ANY(IP)-SAME(PM) OR bus_failed
```

Given the sets:

```
OP:{out1, out2}, IP:{in1, in2}, PM:{x, y}
```

Result of GOD parse (set instantiation):

```
Omission-out1-x = Omission-ANY(IP)-SAME(PM) OR bus_failed
Omission-out1-y = Omission-ANY(IP)-SAME(PM) OR bus_failed
Omission-out2-x = Omission-ANY(IP)-SAME(PM) OR bus_failed
Omission-out2-y = Omission-ANY(IP)-SAME(PM) OR bus_failed
```

Result of subsequent GID parse (operator expansion):

```
Omission-out1-x = Omission-in1-x OR Omission-in2-x OR bus_failed
Omission-out1-y = Omission-in1-y OR Omission-in2-y OR bus_failed
Omission-out2-x = Omission-in1-x OR Omission-in2-x OR bus_failed
Omission-out2-y = Omission-in1-y OR Omission-in2-y OR bus_failed
```

This instantiation process is automated within the HiP-HOPS implementation[5], essentially making a seamless transition from generalised expression to analysis results.

3.3 GFL in practice

GFL increases the potential for reuse, but without a managed approach to performing reuse, it does not improve much upon the current HiP-HOPS method. To successfully apply the GFL with the aim to reuse, a set of GFEs should be well documented and defined as a *pattern*. Therefore this section develops methods for ensuring appropriate capture and reuse of patterns, and also provides some examples of such patterns.

3.3.1 Templates: Documenting Patterns

Patterns must be properly documented with information about their purpose and application. This will still ultimately rely on the judgement of the user to interpret the documentation correctly, but the aim is to provide as much help as possible to ensure safe and appropriate reuse. In this respect, the goals are similar to those of software

[5] Implementation details can be found in the next chapter.

engineering design patterns; GFL patterns are described and documented in a consistent way via the use of templates. Fulfilling the requirements of the template mean the following information is included within a pattern:

- The name of the pattern
- A text description of the abstract behaviour the pattern represents
- Guidance on the implementation/use of the pattern
- The set of generalised expressions for the behaviour
- Any inherited or overridden values (if this pattern is a specialisation)
- Any values for the abstract sets needed to implement the solution/instantiate the pattern (e.g. what failure classes apply)
- Examples of instantiated pattern(s)
- Cross references to related patterns, or patterns that could be used in conjunction with the described pattern

The name of the pattern should clearly and succinctly identify what the pattern represents, as it is the first step to selecting a pattern. Examples would be ‘Propagator’ or ‘Majority Voter’. The name should be kept as concise as possible, as the main detail is kept under the next heading, a specific description of the behaviour represented in the pattern. This should explain precisely what behaviour is captured by the pattern. In most cases however, there is a need to explain in more detail the specific circumstances for the pattern, thus included is a section for usage notes. Including these notes should clear any ambiguity about the pattern; it should be as long as deemed necessary to fully explain the pattern.

The template should provide all the necessary documentation to guide a pattern’s application. The detail of the template is for defining information used for annotation and parsing by the tool or indeed for the user to study to gain a detailed insight of the pattern. This will include the full set of behaviour expressions, most likely in GFL format. Where inheritance is used, the declaration of inherited behaviour is given along with any overridden expressions. Next, applicable values for sets can be specified if the values themselves are general and do not need to be taken from the model (for example standard failure classes). Examples of instantiated expressions are given as a reference to ensure correctness. Finally, a list of related patterns is included mainly for the analyst as a provision to ensure correct pattern selection, as there may be times when a similar but more appropriate pattern may be available and should be considered. The following pages provide some example patterns in fully-documented form:

PATTERN	Direct Propagation
DESCRIPTION	Propagates any failure at any input to every output.
USAGE NOTES	Designed for use with normal failure classes such as omission, value, commission, etc. and data-based input/output parameters. Can work with any number of input/output ports.
FAILURE LOGIC	FC-OP-PM = SAME(FC)-ANY(IP)-SAME(PM)
FAILURE CLASSES	Omission, Value, Commission, Timing

INSTANTIATED EXAMPLES	Omission-out-signal = Omission-in-signal
	
	Value-out-signal = Value-in-signal

RELATED PATTERNS	Multiplexer, Demultiplexer, Propagator Bus

Table 4 Direct Propagation pattern

In the Direct Propagation pattern, any failure of any input parameter will cause the same parameter failure at the output. This type of behaviour is found for example in bus components and multiplexers/demultiplexers.

PATTERN	Global failure in the same mode
DESCRIPTION	Omission of every output is caused by a single common cause internal basic event.
USAGE NOTES	Designed for data-based input/output parameter. Only omission failure classes are compatible. Can work with any number of output ports/parameters.
FAILURE LOGIC	Omission-OP-PM = InternalFailure
FAILURE CLASSES	Omission

INSTANTIATED EXAMPLES	Omission-out-signal = InternalFailure
	
	Omission-monitor = InternalFailure

RELATED PATTERNS	Multiplexer, Demultiplexer, Propagator Bus

Table 5 Global Failure pattern

Global failure is used where an omission of each output parameter at each port is caused by a single failure mode of the component, for example an internal malfunction of a component resulting in a loss (omission) of all outputs.

PATTERN	Fail Silent (Transformation of Failure)
DESCRIPTION	Any input failure class is transformed into an omission of the output.
USAGE NOTES	Designed for data-based input/output. Only omission failure classes are compatible. Can work with any number of output ports.
FAILURE LOGIC	Omission-OP = ANY(FC)-ANY(IP)
FAILURE CLASSES	Omission, Value, Commission
INSTANTIATED EXAMPLES	Omission-out = Value-in1 OR Value-in2 OR Commission-in1 OR Commission-in2
RELATED PATTERNS	Common-cause failure, Bus failure

Table 6 Fail Silent pattern

This pattern is for components that can transform detected input failures into another type, in effect failing silently. For example, value or timing failures could be transformed into omissions. In another scenario, value failures for instance could be propagated, as by their nature are difficult to detect, whereas other types such as timing or commission failures are detected and transformed into omissions. Parameters are not specified as the output is an omission (i.e. there is no signal output).

PATTERN	Standby Recovery
DESCRIPTION	For a standby component that monitors a primary input and activates standby in that case. Both primary and standby must fail for the subsystem to fail.
USAGE NOTES	Standby is assumed to be activated upon omission of output from the primary. Two inputs assumed, monitor (for the primary) and input (for the common input to both components), and a single output assumed.
FAILURE LOGIC	FC-output-PM = O-primary-SAME(PM) AND SAME(FC)-standby-SAME(PM)
FAILURE CLASSES	Omission, Value, Commission, Timing

INSTANTIATED EXAMPLES	Omission-out = Omission-primary AND Omission-standby
	Commission-out = Omission-primary AND Commission-standby

RELATED PATTERNS	Fail Silent, Direct Propagation

Table 7 Standby Recovery pattern

The pattern for a Standby Recovery component captures behaviour where a combination of an omission of the primary input and any other type of failure at the secondary input will result in a failure at the single output of the same type as at the secondary input. This can model a component where a primary input is monitored, and in the case that an omission is detected, the secondary input takes over. Any failures of the secondary input would then be propagated to the output.

PATTERN	Redundancy on Inputs
DESCRIPTION	All inputs must fail to cause a corresponding failure of output.
USAGE NOTES	Can work with any number of input or output ports/parameters.
FAILURE LOGIC	FC-OP-PM = SAME(FC)-ALL(IP)-SAME(PM)
FAILURE CLASSES	Omission, Value, Commission, Timing

INSTANTIATED EXAMPLES	Omission-out-signal = Omission-in1-signal AND Omission-in2-signal
	Commission-out-signal = Commission-in1-signal AND Commission-in2-signal

RELATED PATTERNS	Direct Propagation, Demultiplexer, Multiplexer

Table 8 Redundancy on Inputs pattern

Redundancy on Inputs can be used for components that can tolerate failure of some inputs, continuing to operate correctly until all inputs have suffered a failure.

PATTERN	Majority Voter
DESCRIPTION	All majority of inputs must fail to cause an omission of output
USAGE NOTES	Can work with any number of input port/parameters but assumes a single output. Possible variation assumes same FC instead.
FAILURE LOGIC	Omission-out-PM = ANY(FC)-MAJ(IP)-SAME(PM)
FAILURE CLASSES	Omission, Commission, Value, Timing
INSTANTIATED EXAMPLES	Omission-out = Omission-in1 AND Omission-in2 OR Omission-in1 AND Omission-in3 OR Omission-in2 AND Omission-in3
RELATED PATTERNS	Redundancy on Inputs

Table 9 Majority Voter pattern

The Majority Voter is another type of ‘redundancy’ pattern; here an omission of output (i.e. fails silent) is caused by a common failure occurring across a majority of input ports.

PATTERN	Specialised Bus
DESCRIPTION	Propagates any failure at any input to every output; value failures also can be caused by an internal failure.
USAGE NOTES	Designed for use with normal failure classes such as omission, value, commission, etc. and data-based input/output parameters. Can work with any number of input/output ports.
INHERITS	Direct Propagation *FC-OP-PM = SAME(FC)-ANY(IP)-SAME(PM)*
FAILURE LOGIC	Value-OP-PM = Value-ANY(IP)-SAME(PM) OR InternalFailure
FAILURE CLASSES	Omission, Value, Commission, Timing

INSTANTIATED EXAMPLES	Omission-out-signal = Omission-in-signal
	Value-out-signal = Value-in-signal OR InternalFailure

RELATED PATTERNS	Multiplexer, Demultiplexer, Propagator Bus

Table 10 Specialised Bus pattern

The Specialised Bus pattern above is provided as an example of how inheritance (reuse via specialisation) works within a template. Here, the component would exhibit the behaviour of the Direct Propagation, but with the addition of a further cause for value failures at output. In practice the expressions from inherited patterns do not need to be shown, it is included here for clarity.

As a minimum, patterns of behaviour should be identified, specified according to the given template, and stored appropriately during the course of designing a system. A large store of patterns would eventually be available not just for the system under development, but for the design and analysis of other systems. However, developing methods for managing a library of patterns is not part of this work[6].

3.3.2 Steps for defining behaviour

A difficulty that remains is the problem of choosing the correct pattern to apply in a given scenario. The most important point is that the incorrect pattern selection can lead to incorrect or incomplete analyses. Therefore it is essential now that a structured methodology is defined to guide the decision making process, so that the application and reuse of patterns is correct and consistent. This work hopes to encourage the analyst to study available patterns carefully, to minimise the chance of error. The process of choosing a pattern can be guided with the following general steps:

1. Determine the component’s failure behaviour

The first and most important step is to make sure that it is known what behaviour the component is expressing. Following that, it is possible to make a decision about how this can be represented generally. To do this, consider the effects on all inputs and outputs of the component in conditions of failure. In particular, focus on those deviations of output which are caused by deviations of input, either directly or by some combination. Also consider any internal events generated by the component which affect output. Thinking about and defining the behaviour at the highest level of abstraction should make it easier to search for and relate to the exact or most appropriate pattern.

[6] This is discussed in section 5.2 Further Work.

2. Read the pattern's description

The available pattern library must be consulted for appropriate candidate patterns. A pattern's documentation must then be consulted to ensure the correctness of the choice. Important here is that the behavioural features determined in step 1 must be contained within the detail of the given pattern description. If the documentation suggests otherwise, other patterns should be considered until the most appropriate is found. If there is no suitable pattern available, the situation would imply that a new pattern should be created to capture this component's behaviour. The remaining steps should still be taken before creating a new pattern, as it may be possible to combine existing patterns to reach the desired description. Even if creating a new pattern, the remaining steps are still valid, but with a change of purpose (creating rather than choosing a pattern).

3. Look at related patterns

Included in the documentation of every pattern is a list of related patterns; other behavioural types similar or within the same class as the chosen pattern. The purpose of this list is to direct the search towards the right pattern in cases where the chosen pattern may not be entirely suitable. The descriptions of related patterns should always be read before the selection is finalised, as it is important to use the most suitable pattern available.

4. Choose the most abstract pattern

Some patterns may contain expressions which appear to be very abstract and be difficult to interpret. In some cases there may be a pattern which contains a less abstract definition, one which is simpler for the analyst to understand by reading the expression, which will fit the requirements. Where possible, the most abstract pattern should be chosen that can still represent the necessary behaviour. The reason is that once annotated, it will allow a greater scope for reuse of the component under analysis. For example if a component is likely to undergo a change where the number of inputs is altered, a good choice of pattern would be one that includes expressions with an abstract term for input ports.

5. Make sure all behaviour has a definition

The final step is to check that all behaviour identified in step 1 has been successfully represented within a pattern. This can be achieved via instantiation of the pattern and manual search of expressions generated. Every output deviation as defined in step 1 must be listed in an expression with its input deviation cause(s). Instantiation can be done manually, however tool support can be employed for automation[7].

With a strong library of patterns, these steps to selecting a pattern should be sufficient. However it is possible that even if all of the previous steps to selecting a suitable pattern have been followed, there may still be situations where no appropriate pattern can be found. There are two likely scenarios:

1) A pattern close to what is required, but lacking all the necessary detail

2) No suitable pattern found

In both scenarios, a possible cause is that the required behaviour is available across multiple patterns, but not within the same pattern. The simplest solution in this case is to search for all the individual patterns (starting again at step 1) and then to employ inheritance to capture all the required behaviour into a new pattern. This is the preferred option, but it is also possible to use overriding behaviours if an expression is not specific enough. Use of this approach should be limited, because if many overriding

[7] Automated instantiation is used in validation of the GFE concept (section 4.3).

expressions are needed, a likely better solution would be the creation of a new pattern that specifically includes all of the necessary behaviour.

Even with inheritance, there are still situations where available patterns lack the required detail. The alternative solution is to create a new pattern by manually adding the extra behaviour to an expression. For example, given the base behaviour:

```
Omission-out = Omission-in
```

An extended version can be created by appending the required behaviour, for example:

```
Omission-out = Omission-in OR InternalFailure
```

Clearly, this is a case of trivial reuse however it is still more desirable to use this approach than to create a new pattern that is not based on an existing pattern of well-known behaviour.

This methodology is given only as a guideline and it would be expected in practice analysts would develop their own procedures along these lines. As such this guide does not aim to define the process, rather it tries to promote a structured method on which to base the selection of patterns.

3.3.3 GFL applied to the fuel system case study

Using the extended HiP-HOPS syntax, the previous fuel system case study is now revisited to demonstrate how the GFL can be applied to annotations. Table 11 has the GFL versions of the annotations found in Table 3. A comparison of these tables highlights the benefits of GFL annotation over that of classical HiP-HOPS. For example the computer can have all its behaviour defined with just two expressions because there is no need to create an expression for every output port. Not only does this reduce the workload of the analyst, but the increased level of abstraction also means that if the computer were used in another system or design, the set of GFEs would likely need little or no alteration in order to be compatible, thus more easily reused. Another example is the power bus which previously required three expressions and can now be represented by one shorter general expression. The benefits also extend to the scalability of GFL expressions, such as the power bus expression being valid for any number of input connections, meaning the addition of more PSUs to the system would not result in re-annotation.

PSU
Omission-out = PSU_failed
Power Bus
Omission-OP = Omission-ALL(IP)
Computer

Omission-OP-PM = Omission-power OR computer_failure
Commission-OP-PM = memory_stuck

Data Bus

Omission-OP-PM = Omission-ANY(IP)-SAME(PM) OR bus_failed
Commission-OP-PM = Commission-ANY(IP)-SAME(PM)

Logic

Omission-out-PM = Omission-ALL(IP)-SAME(PM)
Commission-out-PM = Commission-ANY(IP)-SAME(PM)

Valve

Omission-out = Omission-in OR stuck_closed
Commission-out = Commission-in OR stuck_open

Pump

Omission-out = Omission-ANY(IP:{in, control, power}) OR
stuck_closed
Commission-out = Commission-control OR stuck_open

Table 11 Fuel system GFL annotations

3.4 Summary

The investigation in this chapter has focused on component failure behaviour and how it is currently represented within a contemporary safety analysis tool, HiP-HOPS. Many components are designed to perform a function irrespective of the system they are used in, for example a component may propagate a signal without needing to know what the signal means, independently of the application context. Furthermore, there are components that exhibit patterns of behaviour which hold in any context. The HiP-HOPS safety analysis tool is limited with regards to capturing such patterns within its component annotation syntax, limiting the reusability of failure behaviour expressions. Based on these factors, and to meet the thesis aim of improving the scope of reuse within safety analysis, the Generalised Failure Language (GFL) has been developed as an extension to the classical HiP-HOPS component annotation syntax.

Development of the GFL involved breaking down expressions of failure behaviour into their constituent terms and replacing these, where possible, with more abstract, general terms which could represent a set of values, rather than a single specific value. In addition, a series of abstract operators have been added to the syntax which can make collective references to sets of values and enable more general statements of failure behaviour to be captured, such as ‘omission of all inputs’. A small example system was used to demonstrate the practical benefits that GFL can bring to an analysis, particularly the reduction in the annotation effort and the scalability of generalised expressions.

The GFL allows the capture of general behaviour and gives scope to reuse, but it is acknowledged that without further detail and systematic methods, performing reuse appropriately is a challenging task. Addressing this issue, this development has aimed to reduce the likelihood of *ad hoc*, ‘cut & paste’ style reuse by ensuring that reuse is not seen as a shortcut and instead is treated as a way to safely use existing, proven failure information in compatible situations in other systems. This has led the development of the concepts of GFL inheritance and guiding documentation, taking inspiration from software engineering and design patterns. A small set of example patterns has been provided which show the typical kinds of behaviour found in safety critical systems, such as fail-safe redundancy. Inheritance provides the benefit that a component could have a very simple annotation, such as 'Inherit Bus'. The danger is if the stored version of the Bus pattern is changed or lost, so as part of the implementation, behaviour is automatically brought into the local model. This way the annotation can retain its simplicity, but the analysis will still benefit from the reuse.

The final point made was that all the generalisation and guiding documentation introduced is made irrelevant if appropriate procedures are not followed for carrying out reuse, e.g. the process of selecting a pattern. Addressing this issue, a set of guidelines has been provided on how to choose a pattern that is appropriate for a given component, but it remains a manual process and its success depends upon the expertise of the analyst.

4 Application

This chapter demonstrates the application of the Generalised Failure Language (GFL) as proposed in Chapter 3. A working prototype of a GFL-extended HiP-HOPS tool has been developed in order to prove the concept. The discussion in this chapter is focused on the existing processes of HiP-HOPS which are extended as a result of the GFL integration. The new software is demonstrated on a real-world case study of an automotive brake-by-wire system and evaluations are made of the effectiveness and applicability of the GFL with regards to the original research criteria, based on a series of legacy case studies converted into GFL format and compared with classical HiP-HOPS.

4.1 Tools and process

The failure annotation and the model parsing stages of HiP-HOPS are the processes which are the focus of the extended GFL method. During the second step, at the point which HiP-HOPS performs the fault tree synthesis, the current and GFL techniques converge and the remaining analysis process is the same for both methods.

4.1.1 Model annotation

HiP-HOPS supports several commercial modelling tools. In this work, the implementation focuses on HiP-HOPS as used with MATLAB/Simulink (Mathworks, 2009), hence a 'model' refers to a Simulink model.

In a typical model, components and subsystems are represented as 'blocks' in the model diagram, so the topology of the system is displayed as a series of interconnected blocks (see example in Figure 13). To perform a safety analysis on a model, definitions of failure behaviour are first added to blocks manually, requiring the use of an external interface tool which runs in conjunction with MATLAB/Simulink. HiP-HOPS provides such an interface so that blocks can be selected from within the model and annotated with expressions of local failure behaviour[8], as well as other statistical data for further probabilistic analyses. A prototype interface has been developed (shown in Figure 14) to support annotations using the existing or new generalised syntax, or any combination of these within a model.

[8] The reader is referred to section 2.2.5 and section 3.1 for more discussion of HiP-HOPS and the construction of failure behaviour expressions.

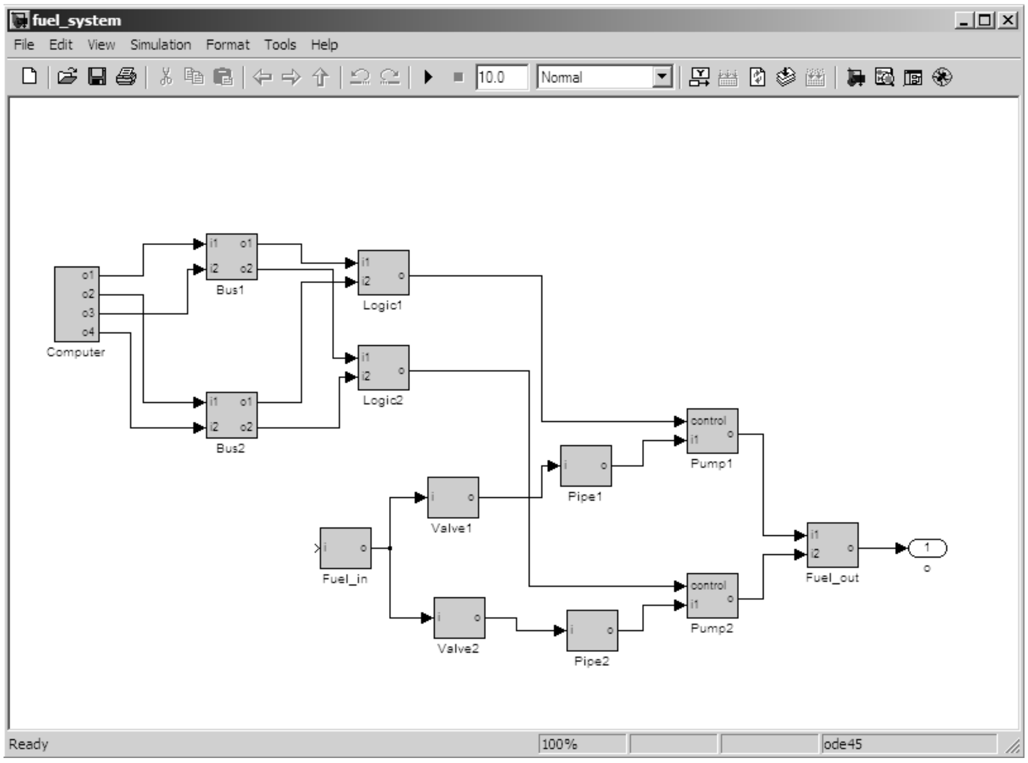
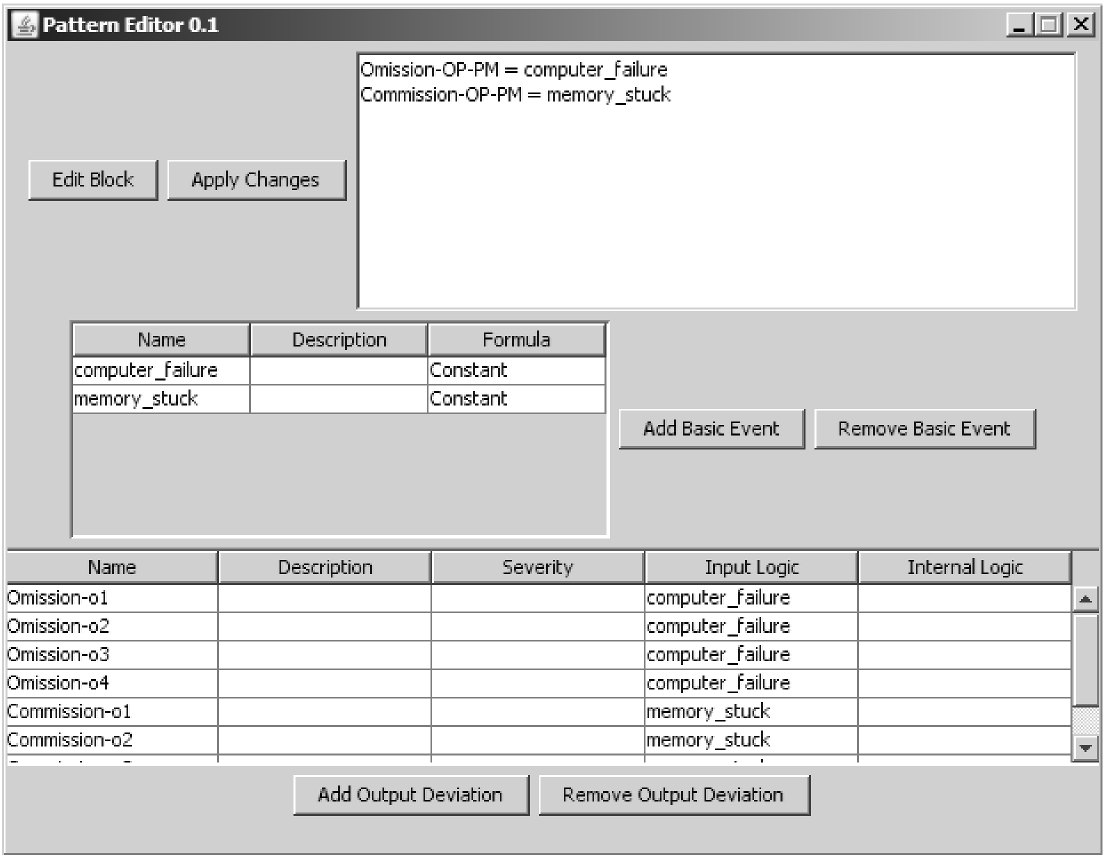


Figure 13 Example MATLAB/Simulink model view of a hypothetical fuel system



Omission-OP-PM = computer_failure
Commission-OP-PM = memory_stuck

Name	Description	Formula
computer_failure		Constant
memory_stuck		Constant

Name	Description	Severity	Input Logic	Internal Logic
Omission-o1			computer_failure	
Omission-o2			computer_failure	
Omission-o3			computer_failure	
Omission-o4			computer_failure	
Commission-o1			memory_stuck	
Commission-o2			memory_stuck	

Figure 14 Prototype GFL annotation interface

4.1.2 Model parsing

The automated stage of an analysis under HiP-HOPS begins with the model being parsed. Part of this process is to check that failure behaviour has been described correctly according to the grammar of HiP-HOPS expressions. Assuming that components have been given appropriate annotations in the model, the information about failures in conjunction with the system topology is used to automatically construct the set of fault trees and FMEAs for the system.

Extending the classical HiP-HOPS annotation syntax with GFL notation requires a modified parsing process. This must be capable of successfully interpreting abstract terms and instantiating ‘concrete’ expressions, given any necessary application-specific values. This section demonstrates with a running example how GFEs can be automatically interpreted and where this fits in the overall analysis process. Consider the structure of the following GFE:

```
FC-OP EXCEPT:{out2}-PM = ANY(FC)-ALL(IP)-SAME(PM)
```

where `FC:{Omission, Commission},OP:{out1, out2},`

```
IP:{in1, in2, in3},PM:{x, y}
```

Additional processing of the extended parser amounts to sets being instantiated and operators being expanded. These processes follow the production rules as described by the GFL grammar so that the correct ‘concrete’ expressions are always obtained from a given pattern. To integrate the GFL successfully with HiP-HOPS, any GFEs are parsed and transformed into classical HiP-HOPS expression format, which are then used as input to HiP-HOPS's fault tree synthesiser for the remaining analysis steps[9]. It is during this phase that HiP-HOPS can automatically retrieve necessary application-specific information from the model (i.e. references to failure classes, ports and parameters) and substitute them for abstract terms where necessary. This way the GFL concept can be based upon the established HiP-HOPS fault tree synthesis method, allowing comparison and validation of fault trees generated from GFEs against those generated from legacy annotations.

To parse a GFE, each side of the expression (output deviation (OD) and input deviation (ID)) must first be considered separately. The first step is to instantiate every possible OD from any abstract sets in the OD terms. This proceeds in the order FC > OP > PM; terms in an OD are parsed left-to-right. Each set is iterated through and every combination of values instantiates an OD. For example, the GFE above with the given values contained in each set would initially be instantiated into four ODs, each sharing the initial generalised ID:

```
 Omission-out1-x = ANY(FC)-ALL(IP)-SAME(PM)

 Omission-out1-y = ANY(FC)-ALL(IP)-SAME(PM)

Commission-out1-x = ANY(FC)-ALL(IP)-SAME(PM)

Commission-out1-y = ANY(FC)-ALL(IP)-SAME(PM)
```

At this stage, if an INHERIT directive is specified in the pattern, any necessary behaviour is brought into the annotation from the library of available patterns. Also, if inheritance-overriding or EXCEPT on an output deviation term are used, this is the earliest opportunity for any of these 'exclusions' can be carried out as required by the

[9] This approach was used to create the prototype of the extended HiP-HOPS tool presented in this thesis. Further development may render this step unnecessary.

pattern. For example, overriding a behaviour implies that an expression is made redundant and can be eliminated at this stage, similarly EXCEPT will reduce the amount of individual values to be processed; thus making the 'exclusions' at an early stage can reduce the overall workload of the parser. In the above expressions, the term `out2` has already been subject to an EXCEPT operation and has no further consideration in the remaining parse. Once the OD parse is complete, the set of expressions obtained will be the total number of expressions that can be produced by the GFE, as no more expressions can be added during the ID parse.

The next step therefore is the expansion of the remaining generalised ID terms for each expression in the set. The first expansion necessary is to replace any instances of SAME with the appropriate value taken from the instantiated OD side. This is done first in order to improve efficiency, as expansion of other operators increases the number of instances which would need replacing later (as can be inferred from later steps in this example). Thus, continuing with the above example, expanding the SAME terms gives:

```
 Omission-out1-x = ANY(FC)-ALL(IP)-x
 Omission-out1-y = ANY(FC)-ALL(IP)-y
Commission-out1-x = ANY(FC)-ALL(IP)-x
Commission-out1-y = ANY(FC)-ALL(IP)-y
```

The second step of ID expansion is to process any remaining operators, ANY, ALL and MAJ; again any set values excluded with EXCEPT are ignored from the parse. It is important to note that these operators do not decide the precedence in which terms in an ID are parsed, indeed parsing of IDs uses the same ordering as for ODs, i.e. FC > IP > PM. However, the resulting expanded expression must still obey standard Boolean precedence rules otherwise an expanded GFE can take a different meaning to its intended behaviour representation. For instance it is essential that within an expression, precedence is given to conjunction (AND) over disjunction (OR) for any fault trees to have the correct structure i.e. the combination A AND B OR C should be interpreted as (A AND B) OR C, not as A AND (B OR C), as these expressions represent two different logical behaviours. Thus, continuing with the example, the next expansion is of the `ANY(FC)` term:

```
  Omission-out1-x  = Omission-ALL(IP)-x OR Commission-ALL(IP)-x

  Omission-out1-y  = Omission-ALL(IP)-y OR Commission-ALL(IP)-y

 Commission-out1-x = Omission-ALL(IP)-x OR Commission-ALL(IP)-x

 Commission-out1-y = Omission-ALL(IP)-y OR Commission-ALL(IP)-y
```

Following this, `ALL(IP)` is expanded to give:

```
Omission-out1-x =     Omission-in1-x AND Omission-in2-x AND
                      Omission-in3-x OR Commission-in1-x AND
                      Commission-in2-x AND Commission-in3-x

Omission-out1-y =     Omission-in1-y AND Omission-in2-y AND
                      Omission-in3-y OR Commission-in1-y AND
                      Commission-in2-y AND Commission-in3-y

Commission-out1-x =   Omission-in1-x AND Omission-in2-x AND
                      Omission-in3-x OR Commission-in1-x AND
                      Commission-in2-x AND Commission-in3-x

Commission-out1-y =   Omission-in1-y AND Omission-in2-y AND
                      Omission-in3-y OR Commission-in1-y AND
                      Commission-in2-y AND Commission-in3-y
```

Once every abstract term in every expression has been instantiated and expanded, the full set of expressions is passed back to the fault tree synthesiser and the final processing continues as per the classical HiP-HOPS process.

4.2 Case study: brake-by-wire system

The brake-by-wire (BBW) system is a design for applications in the automotive industry, developed by Daimler Research as part of the Time-Triggered Architectures (TTA) (Heiner and Thurner, 1998) project[10]. Figure 15 gives the general design of the system. The intended purpose of this architecture is to replace the traditionally hydraulic control of a vehicle's braking system with a more sophisticated electronic control, leading towards more advanced operation and hence improved safety features. This case study examines a simplified version of this system as a practical application of the GFL-extended HiP-HOPS.

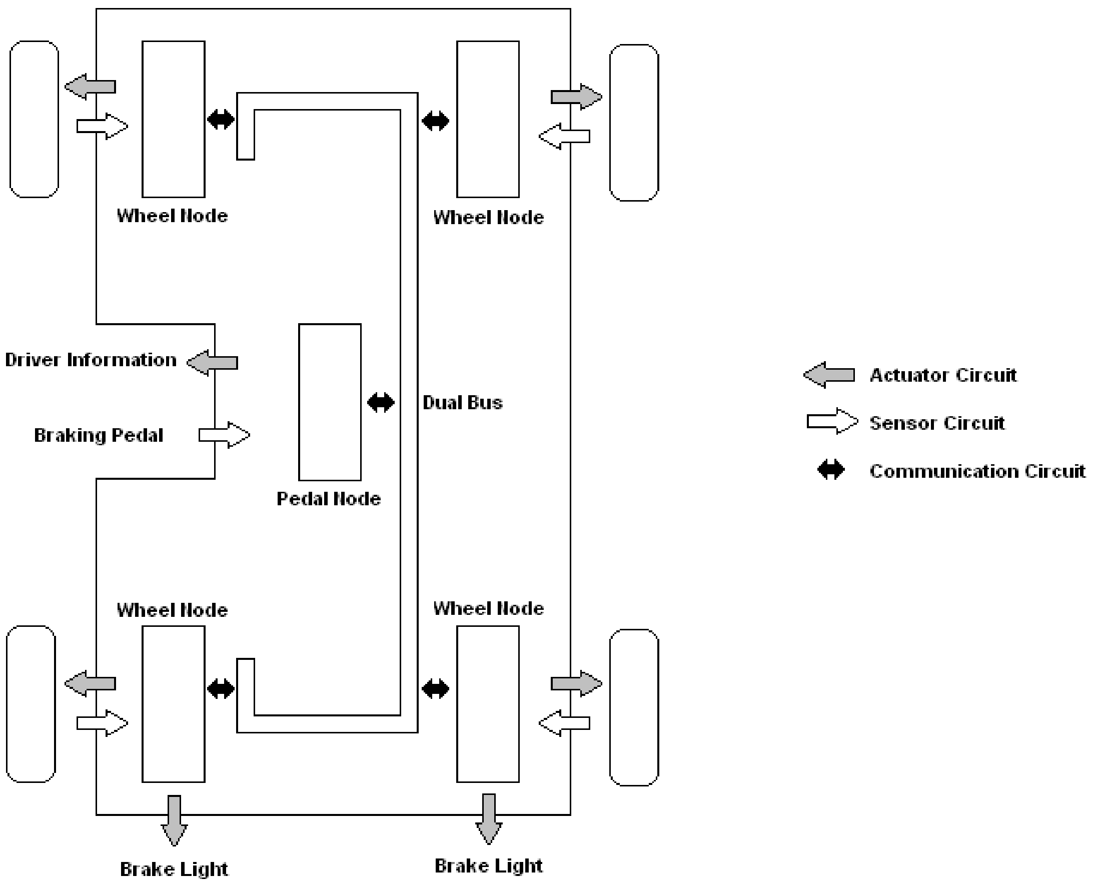


Figure 15 The brake-by-wire system architecture (Papadopoulos *et al.*, 2001)

[10] European Commission funded ESPRIT project 23396 (Papadopoulos *et al.*, 2001).

The brake-by-wire system is constructed as a network of programmable nodes that communicate a braking signal. The brake pedal is continuously monitored for the signal to be initiated. When the brake pedal is depressed, the pedal node receives and broadcasts the braking signal over the network on two replicated buses (a fail-safe measure). The wheel nodes receive the signal and in response, activate the braking actuators.

The provision of an electronic control enables further information such as the load on the wheel, the rotational acceleration and other sensory feedback to also be considered by the wheel node. Thus the aim of the system is to provide a more accurate and appropriate measure of the braking pressure required in the current state of the vehicle. Overall, more sophisticated management of the braking functions is provided, such as braking proportional to each wheel's load, in addition to anti-lock braking and electronic stability control. The Simulink model of the brake-by-wire system used in this study can be seen in Figure 16.

All communications in the node network are done over the time-triggered communications protocol, TTP/C (Kopetz *et al.*, 1989). The TTP/C controller component handles the communication of signals and is designed for fault-tolerance in the event of a failure. For example, in a time-triggered network where signals must be delivered to a predefined schedule, transforming the hazardous failure types such as timing failures into omission failures means the component will fail-silent. An abstract diagram of the TTP/C failure behaviour can be seen in Figure 17, which shows how the different failure types are handled and propagated by the controller network.

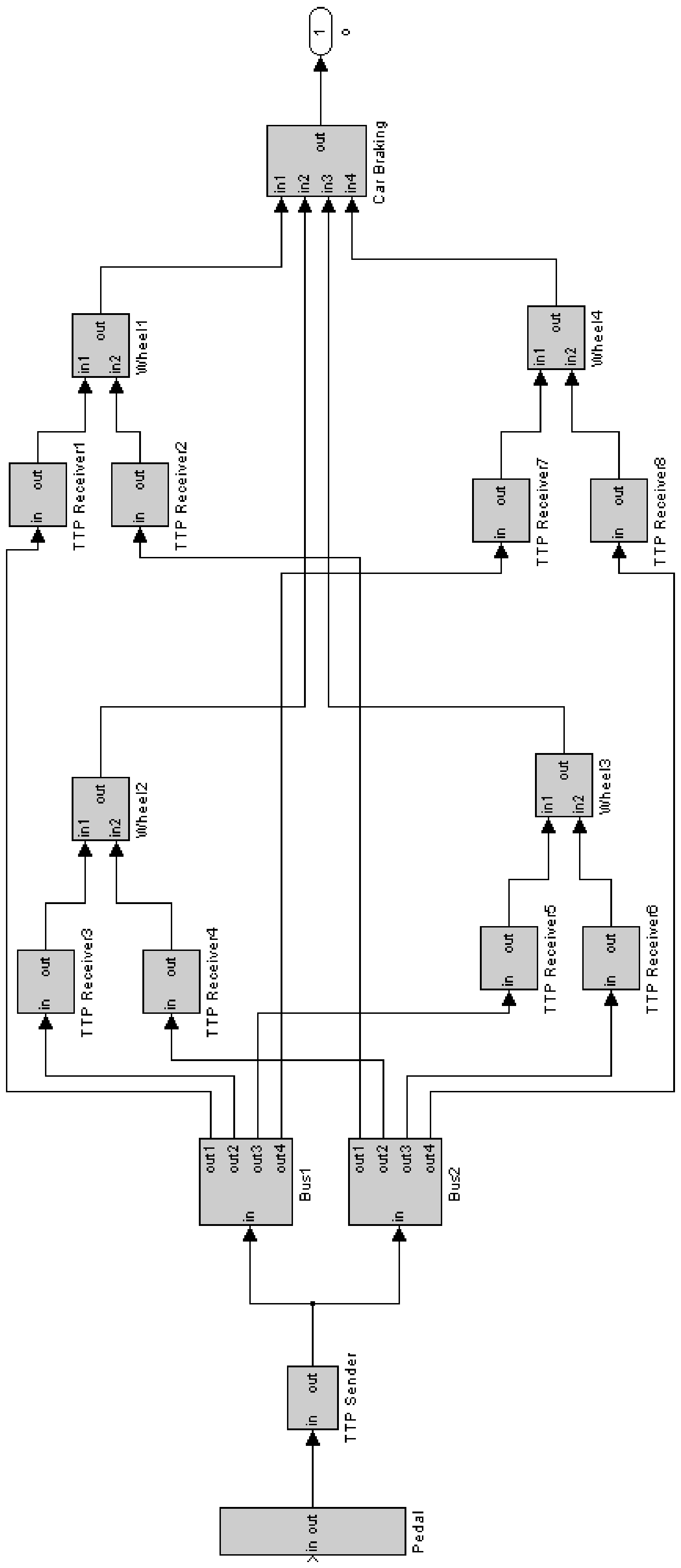


Figure 16 Simulink model of the brake-by-wire system

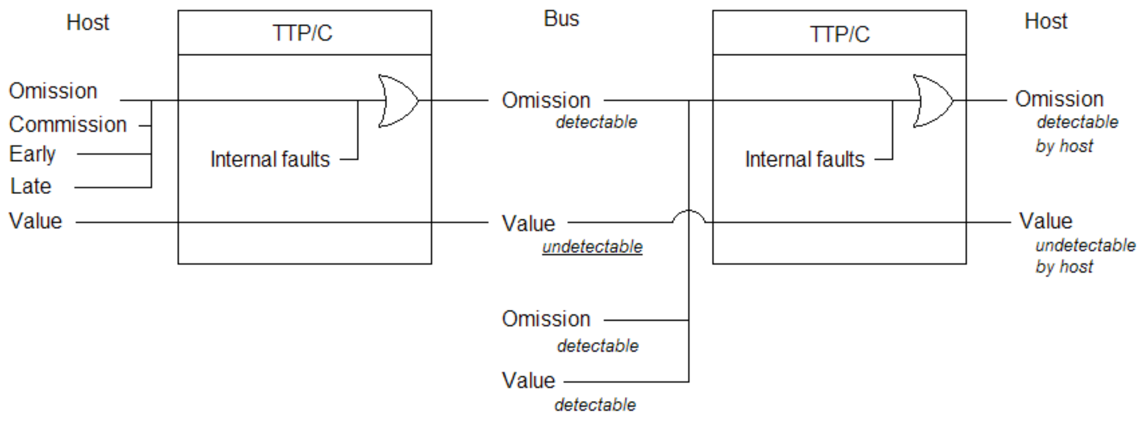


Figure 17 TTP/C failure behaviour (Kopetz and Grünsteidl, 1994)

4.2.1 Analysis of the brake-by-wire system

The safety assessment begins with a detailed investigation of each component's failure behaviour. The required information for the tool comes from an examination of deviations across component interconnections, as per the standard HiP-HOPS process. Where possible, behaviour is described using generalised terms to create the set of GFEs for each component, which are then be annotated to the blocks in the model. The expressions for each component are given in Table 12, however in the course of this case study the expressions have been simplified by assuming there is only one parameter associated with each port, the braking signal, and therefore abstract references to parameter terms are not included.

Pedal
`Omission-out = PedalFailed`
`ValHigh-out = PedalBiasedHigh`
`ValLow-out = PedalBiasedLow`
TTP/C Controller
`Omission-out = ANY(FC EXCEPT:{ValHigh, ValLow})-in OR ControllerFailed`
`FC:{ValHigh, ValLow}-out = SAME(FC)-in`
Bus
`FC EXCEPT:{Val_Detectable, Omission}-OP = SAME(FC)-in`
`Omission-OP = Omission-in OR BusFailed`
`Val_Detectable-OP = EMI`
Wheel
`FC EXCEPT:{Omission}-out = SAME(FC)-ANY(IP)`
`Omission-out = Omission-ALL(IP) OR BrakeFailed`
Car Braking
`NoBrake1-out = Omission-ANY(IP)`
`NoBrake3-out = Omission-MAJ(IP)`
`NoBrake4-out = Omission-ALL(IP)`
`NoBrakeFront-out = Omission-FrontLeft AND Omission-FrontRight`
`NoBrakeRear-out = Omission-RearLeft AND Omission-RearRight`
`NoBrakeDiag-out = (Omission-FrontRight AND Omission-RearLeft) OR (Omission-FrontLeft AND Omission-RearRight)`
`ValHigh4-out = ValHigh-ALL(IP)`
`ValLow4-out = ValLow-ALL(IP)`

Table 12 Brake-by-wire component annotations

The simplest component in the system is the brake pedal which has only three possible output deviations: an omission of output caused by an internal failure of the pedal, or a value deviation (high or low) caused by a corresponding bias in the pedal. Commission failures (undesired braking signals), for the purpose of this case study, are equivalent to value deviations where a value of zero was expected (i.e. the value is nonzero). This behaviour is relatively simple and does not contain any terms that can be generalised, so is expressed using the classic syntax.

A virtual component ‘Car Braking’ is used in the model to represent the braking function of the car as connected to all four wheel brakes. Output failures of the car braking are thus system-level brake failures the car can experience, such as single or multiple wheel brake failures. The only value failures affecting the system are those which encompass all four wheels, because it is not possible in this model for an individual wheel to experience a value failure. Brake pedal bias causes the high or low value failures in each wheel and so acts as a single common cause for each type of value failure. The complex failure modes of this component can be concisely represented with a set of GFEs, using the available operators to significantly reduce the length of the expressions.

As can be seen in the Figure 16 model, the remaining components in this system are replicated to some degree; the Wheel nodes, TTP communications and Buses. This replication in the design represents literal component reuse within the same context, meaning identical architecture providing the same functions and, importantly for the analysis, exhibiting the same behaviour in conditions of failure with each instance of the component. Establishing patterns for these components can therefore make the annotation phase more efficient as there is no need to fully re-annotate for each instance.

Signals from the Pedal are communicated to the braking subsystem via two busses, replicated for fail-safe redundancy. Each Bus propagates any failure received at its input to all of its outputs. Omission-type failures can also be caused by an internal failure of the Bus, whereas detectable value failures can only be generated if the Bus is subject to electromagnetic interference (EMI). Each Bus can therefore use the same set of GFEs, which closely follows the Direct Propagation pattern, albeit with some modification, having two failure classes (detectable value and omission) listed as exceptions.

The TTP/C controllers can act as a sender or receiver of signals; in both cases they propagate value failures while transforming any other class of failure into omissions of output. This type of transformation represents the fail-silent property of the TTP/C design, however the component will also omit output if it suffers an internal failure. Importantly the TTP/C controller is designed to express its fail-silent behaviour in any application, and can have the same set of GFEs for both sender and receiver functions. In this case a variation of the Fail Silent pattern can be extended to capture this general behaviour by including the 'ControllerFailed' internal event to the causes of an omission failure. The detectable value failure must also be included as a separate definition, because this class of failure is propagated rather than transformed.

The wheel brakes will fail to deliver the braking pressure to the wheel, an omission of braking, if the braking signal is omitted at all its inputs (i.e. both TTP/C receivers omit output), or if the brake suffers an internal failure. Insufficient or excess braking pressure will be applied if there is a value failure at any input. The wheel brakes therefore exhibit a variation of the Redundancy on Inputs pattern, whereby an omission of output is caused by an omission of all inputs, in this case with the additional cause of 'BrakeFailed'.

The details of each possible braking failure the system can experience are given in Table 13, along with severity levels described with the standard severity classes of (IEC 61508, 1997); catastrophic, critical, marginal and insignificant.

Failure	Effect	Severity	Notes
NoBrake1	Loss of braking in single wheel	Critical	Marginal unless car is braking while on a curved trajectory, in which case it may drift off course
NoBrake3	Loss of braking in three wheels	Catastrophic	80% loss of braking function
NoBrake4	Loss of braking in all wheels	Catastrophic	100% loss of braking function
NoBrakeFront	Loss of braking in front wheels	Catastrophic	65% loss of braking function, 30% loss of car stability
NoBrakeRear	Loss of braking in rear wheels	Critical	35% loss of braking function, 30% loss of car stability
NoBrakeDiag	Loss of braking in diagonally opposite wheels	Critical	50% loss of braking, 15% loss of stability, 15% loss of steering
ValHigh4	Excess braking in all wheels	Critical	Excess braking can lead to loss of steerability if wheels lock, but stability is mostly maintained
ValLow4	Insufficient braking in all wheels	Catastrophic	Major loss of braking function

Table 13 Effects of failure on the BBW system (Papadopoulos and McDermid, 1999)

The GFL-extended version of HiP-HOPS successfully analysed the BBW model and produced a set of fault trees. A summary of the full analysis results is given in Table 14; this is a simplified FMEA table showing only single points of failure, as there are numerous effects of combinations of failures that HiP-HOPS produces, and for clarity have been omitted.

Component	Failure	Effect	Severity
Pedal	BiasedHigh	NoBrake1	Critical
	BiasedLow	NoBrake3	Catastrophic
	PedalFailed	NoBrake4	Catastrophic
		NoBrakeFront	Catastrophic
		NoBrakeRear	Critical
		NoBrakeDiag	Critical
		ValHigh4	Critical
		ValLow4	Catastrophic
TTP/C Sender	ControllerFailed	NoBrake1	Critical
		NoBrake3	Catastrophic
		NoBrake4	Catastrophic
		NoBrakeFront	Catastrophic
		NoBrakeRear	Critical
		NoBrakeDiag	Critical
Wheel_FrontLeft	BrakeFailed	NoBrake1	Critical
Wheel_FrontRight	BrakeFailed	NoBrake1	Critical
Wheel_RearLeft	BrakeFailed	NoBrake1	Critical
Wheel_RearRight	BrakeFailed	NoBrake1	Critical

Table 14 Simplified FMEA showing only single points of failure in the model

The analysis results show that the duplication of the buses and TTP/C receivers has ensured that these components cannot be a single point of failure, hence they do not appear in this simplified FMEA. A brake failure can only derive from these components if both buses, both receivers at the wheel or one bus and one receiver fail in combination at any given wheel. It can also be seen that there is no protection from value/commission failures and that the TTP/C sender is a single point of failure. The sender could be replicated to solve the latter problem, allowing some provision of backup redundancy in the event one sender fails. Solving the value/commission problem requires a different strategy. Incorrect or undesired braking pressure to some but not all wheels is a more severe failure than the same failure occurring in all wheels, as it can lead to a greater loss of car stability. For example the car could veer off course if braking is applied unevenly to the wheels. Though it is theoretically unnecessary to protect against such value/commission failures, as they are propagated from the pedal to all wheels, the nature of such electronic controls means it is still possible to experience individual wheel brake failures (unlike for instance a hydraulic system). Thus the design could still make use of improved detection, for example allowing the wheel to make use of replicated inputs and filtering out value errors via a majority voting procedure.

The design of the BBW system has several elements which are reused, most prominently being the replicated wheel nodes and TTP controllers for the signal communications. The wheel nodes for example are literally reused components and have identical failure behaviour within this application. It is feasible then to define GFEs and create a pattern for a wheel so that all instances can use the established behaviour, enabling vertical reuse of the wheel pattern. The BBW model contains 141 annotated components, which under a classical HiP-HOPS analysis requires a system-wide total of 1297 failure behaviour expressions. Using the GFL and patterns, the number of annotations can be reduced to 714, a 45% reduction in the total.

The common behaviour for the reused components gives a significant improvement to the efficiency of the analysis. For instance the design includes nine TTP controllers, all sharing the same failure behaviour (in both sender and receiver modes), that can be quickly annotated to the model once defined as a TTP behaviour pattern. If the classical HiP-HOPS syntax were used, each TTP controller annotation would require a set of three expressions:

```
Omission-out = Omission-in OR Commission-in OR
               Early-in OR Late-in OR ControllerFailed
ValHigh-out = ValHigh-in
ValLow-out = ValLow-in
```

When expressed generally and defined as a pattern, each instance of the controller can be annotated with ‘INHERIT TTP Controller’, a much simpler definition. This type of efficiency improvement was also seen with the Wheel nodes and Buses, though there were fewer repeated instance.

In terms of the lifecycle, this approach to annotating identical replicated components means that where any changes are made that affect the failure behaviour, the GFEs can be quickly updated and the updated expressions will be automatically applied to each instance. If used carefully, ensuring that affecting changes are genuinely applicable to all instances, this can be a significant efficiency gain for all future analyses of the system.

In the course of the analysis, several components were found to exhibit patterns of failure behaviour which were variations on the example patterns given in section 3.3.1. Where these components were reused in the model, it was possible to apply the patterns and therefore make the annotation process more efficient.

4.3 Validation: Comparison with HiP-HOPS

The BBW case study demonstrated some efficiency gains while reducing the overall annotation burden, with more concise definitions and reducing the overall number of expressions needed. in this section the investigation is expanded by studying a series of legacy models that have been used in prior HiP-HOPS analyses to find a quantifiable benefit of potential efficiency gains in other models and analyses. Comparisons are made according to the following criteria:

- Reduction in the number of expressions
- Reduction of the size/length of expressions

Exploring the number of replicated (identical) components in a design to compare the scope for reusable expressions would not yield particularly meaningful results, because classical annotations could be reused in the same manner as GFEs, i.e. there is no difference between the two methods in terms of reusability.

Furthermore, measuring the overall reusability of GFL expressions would involve investigating how often a particular GFE could be applied within and/or across models. However due to the nature of available models, it was not possible to reliably test this factor. Part of the reuse challenge is one of lexical consistency in the way models are annotated. For example, the failure class ‘omission’ could be described as ‘omitted’ or ‘none’, or other similar names. Models used for this study were taken from a variety of sources and do show these kinds of differences. Therefore the algorithms developed to find patterns within a model cannot accurately compare patterns across models due to these inconsistencies. This type of problem could be solved by demanding consistency from the analyst, but it is unenforceable. It would likely be considered restrictive, and it could be argued that useful detail may be lost. This is partly the reason that templates for GFEs contain a section for typical failure classes that can be used, the aim being to encourage the use of the generic types or classes of failures. Indeed, the naming idiom that leads to quick annotation will, if used in this manner, intrinsically reuse the same failure classifications.

4.3.1 Generalisation of expressions in legacy models

An algorithm has been developed to aid the validation process which can automatically make generalisations of a legacy model's existing annotations. Using this, it is possible to make a determination of how effective GFEs are in reducing annotation effort and further to get an indication of how typically patterns of behaviour occur within systems. The algorithm can reduce the number of expressions and also the size of expressions, though it may be possible that no generalisations can be made within a particular model, and thus no reduction be made.

The algorithm works by identifying those characteristics and structures in expressions which can be abstracted and contracted with the GFE terms and operators. The algorithm is most effective for components that have multiple expressions to define their behaviour. The conversion of classical expressions to GFEs proceeds with the following steps:

- Step 1: Re-order the set of expressions

The first step can be considered as a preliminary pre-processing of the expression set. This is to group together common output terms, for example all expressions referring to the same output port would be grouped together, then within those groups, expressions would be ordered for failure class types and finally once again for any common parameters. The result should be a list of expressions which are in a more suitable order for the subsequent steps.

- Step 2: Reduce the number of expressions

This step detects where the SAME operator can be applied within an expression, based on common values found across a set of expressions. For example the following two expressions:

```
Omission-out1-x = Omission-in1-x
Omission-out1-y = Omission-in1-y
```

can be reduced via SAME to become:

```
Omission-out1-PM = Omission-in1-SAME(PM)
```

The SAME operator can only apply when there is more than one expression for a given component; if the annotation has only one expression, this step can be skipped. The aim of checking for instances of SAME before the contraction with operators (Step 4) is to reduce the overall processing i.e. reducing number of expressions that must be checked for contraction.

- Step 3: Rearrange input deviation terms

The purpose of this step is to sort and group together common logical combinations i.e. conjunctions and disjunctions, where allowable within rules of Boolean logic, so that the final contraction step can proceed more quickly while remaining accurate. This rearranges the right-hand side of an expression in accordance with the precedence rules, for example, the input deviation:

```
Omission-in1 AND (Commission-in2 OR Commission-in3) AND Omission-in4
```

would be rearranged to give:

```
Omission-in1 AND Omission-in4 AND (Commission-in2 OR Commission-in3)
```

- Step 4: Contract input deviation terms

The final step detects where expressions can be contracted with the use of the remaining operators, reducing the overall size or length of an expression and making the final generalisations. This initially checks where ANY or ALL could be applied, for example, the input deviation:

```
Omission-in1 AND Omission-in4 AND (Commission-in2 OR Commission-in3)
```

would be contracted to:

```
Omission-ALL(IP:{in1,in4}) AND (Commission-ANY(IP:{in2,in3}))
```

If an expression has a certain structure, and the required distribution and logical combinations of values, it can be possible to make further contraction for the MAJ operator. For example the input deviation:

```
Omission-ALL(IP:{in1, in2}) OR
Omission-ALL(IP:{in1, in3}) OR
Omission-ALL(IP:{in2, in3})
```

can be contracted to:

```
Omission-MAJ(IP:{in1, in2, in3})
```

4.3.2 Validation Results

The validation study of legacy models showed that in terms of reducing the length of expressions via GFL, the overall reduction is mostly insignificant. For example, the models used in the study had typically fewer than ten expressions that could be subject to any significant reduction in length. This is simply due to there being very few instances where the operators could be applied. Conversely, there were many instances where the overall number of expressions could be reduced. Table 15 lists the results from applying the generalisation algorithm. The example models used in this study were provided by Germanischer Lloyd, Daimler AG[11] and Volvo. They include:

A) A ship engine cooling system[12]

B&C) Two blow-out prevention systems used in offshore platforms[13]

D) A fuelling system[14]

E) An automotive brake-by-wire system[15] (as used in prior case study)

F) A steer-by-wire system[16]

Model	Number of annotated components	Total expressions	Average number of expressions per component	Reduction in expressions via generalised annotation	New total expressions	Average number of expressions per component after reduction	Reduction Ratio
A	109	776	7.1	314	462	4.2	41%
B	20	75	3.7	26	49	2.4	35%
C	27	100	3.7	33	67	2.5	33%
D	65	175	2.7	60	115	1.8	34%
E	141	1297	9.2	583	714	5.1	45%
F	85	204	2.4	12	192	2.2	6%

Table 15 Validation results

The abstract nature of the GFL provides the greatest benefit when applied to those models that have been fully annotated with expressions that capture the complete range of failure behaviour. In such models, the large degree of duplication between multiple expressions can be reduced by using a smaller number of more abstract GFEs. Without

[11] Formerly DaimlerChrysler
[12] (Uhlig *et al.*, 2007)
[13] (Hamann *et al.*, 2008)
[14] (Papadopoulos and Petersen, 2003)
[15] (Papadopoulos and McDermid, 1999)
[16] (Papadopoulos and Grante, 2005)

generalisation, the classical HiP-HOPS syntax often leads to a degree of repetition in the expressions and terms, and is therefore a good example of a situation where the GFL can have a significant benefit.

The study has shown that most models receive a significant reduction in the number of expressions needed. This reduction ratio for most models is in excess of 33%, meaning in most cases at least a third of expressions could be removed via the use of GFL annotation. A significant result is for the steer-by-wire system (model F), which only experienced a 6% reduction of expressions. This is a model which is annotated only for early development stage functional analysis, which uses less detailed annotation and considers a smaller number of input and output ports, in addition to only annotating for omission and commission failures. As such, the components in the model only contain on average 2.4 expressions and consequently provide far less scope for reduction via generalisation.

Looking at the brake-by-wire system (model E), there is a 45% reduction, the highest level of all the examples in the study. This model contains a relatively high average number of expressions per component, which provides the greater scope for reduction. It can also be linked to the fact that the expressions mostly share a similar logical structure, making it an ideal candidate for GFL annotation. A manual inspection of this model revealed that the algorithm had successfully elicited a pattern for a central controller bus. This pattern effectively reduced a large number of expressions by generalising the behaviour: every omission delay or corruption of input messages propagates to the outputs, any bus malfunction causes omission of all messages, EMI during communication causes detectable corruption of all messages. Further inspection of the remaining models revealed that fail silence and fault tolerance patterns often occurred in the failure annotations created by analysts.

The results as shown give an indication of the success of the GFL, however the algorithm used to perform the reductions was a developmental prototype and may not have detected all possible applications, thus results may be an underestimation of the true extent of possible reduction. For example the algorithm does not detect complex patterns consisting of more than one GFE, patterns that incorporate EXCEPT clauses or any that use inheritance or overridden values. As such it is likely that in practical applications the annotation burden would be further reduced. This statement is supported by the fact that this validation study has worked only via reverse engineering of the models to process those expressions already present; if the GFL were used during the creation of the model, it would likely be far easier for the analyst to identify applications for patterns. Furthermore, the GFL is designed to operate alongside a library of well-known patterns; with access to such a resource, the extent of reuse would certainly be improved.

4.4 Summary and Discussion

The introduction of abstract representation within failure behaviour expressions means that it must be investigated how such expressions will be interpreted and to determine whether or not it is possible to implement this as a useable language. This chapter showed that the GFL is implementable, by showing how it can be integrated with an existing contemporary safety analysis tool, HiP-HOPS. The details of all the required steps have been given, along with guides to an implementation and an algorithm for converting legacy expressions into the generalised format. The overall method has been subjected to a process of validation via a comparison with classical HiP-HOPS.

When using the GFL method, the first challenge encountered is the problem of knowing whether or not a component's behaviour (annotation) can be expressed generally or not, or at least to know which expression terms may be represented generally. This would very much depend on the analyst's skill and experience of the GFL. Having a library of patterns available at this stage becomes very useful, though there is still considerable effort needed to firstly create such a library and secondly to search it for an appropriate pattern. If a well-known behaviour is not found in the library, the time and effort spent searching might well be better spent simply annotating the model in the classical syntax. This problem however is not quite as severe as it first appears; assuming a correct, full set of expressions for a given component, constructed in classical HiP-HOPS syntax, it is possible to use the algorithm developed in this chapter to automatically find many generalisations and reconstruct the annotation into a generalised format. The new generalised expressions can then be documented and stored for potential reuse later. Once a pattern has been established, it has been shown to make worthwhile efficiency gains for quick annotation when components are replicated within a system, and for managing and applying changes made in the course of the lifecycle.

Quantifying the benefits of the GFL approach is perhaps the most difficult challenge. What was deemed the most feasible approach, used in this chapter, was to examine and compare the numbers of annotations needed to fully describe behaviour at both the component level and the system as a whole. Doing this gave an indication of how effective GFEs are at lessening the annotation burden and also how often GFEs may be employed successfully within a model. The results of these studies showed that the total number of expressions needed could be reduced by about a third or more compared to using the classical annotation syntax, though it varies considerably depending on the model and component behaviours.

Measuring achievable reuse with this method is difficult and has not been ascertainable with available models. It is feasible, for example, to evaluate how often a particular GFE is employed across various models, but the study in this chapter has been limited by the inconsistency in existing legacy model annotations. Any results gained from the available models in this way would unlikely be indicative of the true scope and scale of achievable reuse. A further significant study would be needed to draw more substantial conclusions.

5 Conclusion

In this thesis, the following hypothesis has been tested:

> "More efficient safety analyses can be achieved via a new approach to *generalising descriptions of failure behaviour*. This approach will allow in practice the more concise specification and limited reuse of *patterns of failure behaviour*"

Testing this required meeting several objectives. Those objectives set out in Chapter 1 are restated below, with a discussion of how each was met.

1. Examine relevant model-based safety analysis techniques to evaluate the current extent of achievable reuse.

The review of classical and contemporary safety analysis techniques in Chapter 2 provided the overall direction of this work. The original, classical techniques for safety analysis were not designed with today's complex systems in mind, and with most modern contemporary methods being based on classical techniques, they tend to carry over some of their inherent limitations. With regards to reusability, it is a major difficulty with most techniques, but this has lead to many advances in contemporary methods which reflect compositional system design within analyses. These have had some success, whereby analysis elements can be 'componentised' and reused, but are still limited by a number of factors, most prominently the necessity for application-specific information for each component.

Thus in order to improve the scope for reuse, expressions of failure behaviour must be made more general so that they can have more than one application. This has been shown to work in the FPTC method, the argument is that there is scope to go further.

Looking at reuse in the wider field of engineering suggested that the overall efficiency of safety assessments can be improved in terms of quick annotation for reused components with a structured method for documenting and performing reuse.

2. Develop a concept for generalised annotation and demonstrate that the proposed concept works by extending an existing safety analysis method to enable specification and reuse of patterns of failure behaviour.

Over the course of Chapter 3, a new language for describing the way components fail has been introduced and developed. The new language was developed as a superset of an existing syntax found in the HiP-HOPS safety analysis tool; this was chosen as a candidate for extension because of the flexibility and extensibility of the existing component annotation syntax. HiP-HOPS is also an advanced and widely published technique that offers a wide range of capabilities, including unique capabilities among safety analysis techniques for design optimisation and allocation of safety requirements (Papadopoulos *et al.*, 2010). Since HiP-HOPS partly defines the state-of-the-art in this area, in this thesis it was considered that useful extensions to the method towards greater generalisation and reuse of failure behaviour would generally benefit research on contemporary work on compositional safety analysis.

With the extended annotation syntax, failure behaviour expressions can now take a more abstract form, describing behaviour in a more general manner than classical HiP-HOPS and other model-based safety analysis methods currently provide. The syntax of the language is designed so that generalised failure expressions (GFEs) can be constructed that remain valid across applications and in subsequent analyses. This extension introduces abstract terms and operators, and in combination with additional language features, forms the main contribution of this thesis, called the Generalised Failure Language (GFL). The implication is that GFEs can be reused along with any component that exhibits a general behaviour in any system, but a larger study of this capability is required before more definitive conclusions can be made. The new approach also improves upon scalability of expressions, for example abstract references can be made to component interconnections (i.e. ports) so that an expression remains valid regardless of the amount of input or output connections the component has, or is modified to have. As a result, describing general behaviour using the newly developed GFL enables the creation of reusable analysis elements, consequently known as patterns.

Examining this new development from a critical perspective, these extensions introduced new problems in the application of the tool, as the more abstract notation of patterns can be difficult to interpret manually. To mitigate this difficulty, mechanisms to perform reuse safely and appropriately, and for constructing and documenting patterns were developed in conjunction with the new language. A template for consistent documentation of patterns was developed and several examples were given. Furthermore an inheritance mechanism was developed and employed to manage the reuse of patterns; this feature also allows the creation of new patterns by a process of specialisation.

3. Evaluate the extended method

To demonstrate the overall concept, Chapter 4 works through an implementation of the new language integrated with HiP-HOPS and demonstrates analysis with the completed tool by performing a case study analysis, and gives further validation via a series of smaller studies. The implementation consisted of a combined parser for the generalised language and algorithm to instantiate patterns. The parser checks correctness while the instantiation is necessary for application-specific expressions to be automatically generated from the abstract pattern.

The main case study of the automotive brake-by-wire system demonstrates how GFEs can replace standard expressions with more concise annotation and how GFL patterns can be found, specified and used as an efficient means to annotate where there are replicated components in the design. This has however only been shown to work where there is literal component reuse within the same model and not across designs.

Validation of the method is somewhat problematic; the difficulty in quantifying improved reusability has been discussed and resulted in the focus being put on validation by comparing the number of annotations required in legacy models compared to the amount needed when using patterns. For a practical demonstration of this, a legacy-expression converter was also developed to allow a retrospective investigation of older models and to evaluate the difference if patterns were applied. Comparing the newly implemented technique with the classical HiP-HOPS method was therefore limited to evaluating the reduction in annotation effort brought about by generalisation. On average, the number of annotations required could be reduced by around a third, but this figure varied enormously depending on the system. However, the case studies performed were severely limited by the availability of suitable examples. Available models were often incomplete and did not contain a full set of component annotations, meaning figures obtained for reduction ratios may not reflect accurately on real-world performance. Nevertheless, the results from the limited study were encouraging with values ranging from less than 10% to over 40% reduction in annotations, and it is reasonable to claim that the reduction ratio in more complex, completely annotated models could be higher than that seen in the course of the study.

5.1 Limitations of the concept

One shortcoming that is expected to create overheads in the application of the concept is the lack of pattern libraries that need to be developed to support this method. Clearly, this additional resource of a library of patterns is needed for the method to achieve its full potential. More importantly, although in theory the proposed GFL enables greater context independence and reuse, ultimately the ability to reuse failure analyses is not dependent on the reusability of the linguistic constructs. Rather, it depends on the sensitivity of these analyses on the context in which the system is being deployed. This context may include different states within the same application or different application environments. This is not an issue that has been investigated in this thesis and little insight has been thrown to this problem which should be further investigated.

There are also some issues relating to the process; these focus mainly on performing reuse with the GFL method. It has some notable limitations which undermine some of the effort made in developing a technique for improving reuse. The inheritance mechanism used at present can be cumbersome and is not as flexible as intended. The original intention was to enable both ‘verbatim’ and ‘trivial’ styles of reuse. Verbatim reuse, where expressions are not modified, is currently the only type possible, albeit with some flexibility with regards to overriding expressions. Trivial reuse of expressions, where only a slight change is made to an expression, is not possible with the proposed mechanism. For example, the TTP/C controller component examined in the case study, though expressing a general fail silent behaviour, could not be annotated with the standard Fail Silent pattern, due to the shortcomings of inheritance. The Fail Silent pattern can still be used as a guide for manual annotation, though a more capable reuse mechanism could allow reuse to take place where extended behaviour is defined within an expression; essentially an inline declaration of inherited values. Further study would be beneficial to investigate the feasibility of this approach.

Another problem with the reuse mechanism in general is that allowing expressions to be overridden when behaviour is inherited brings about opportunities for new kinds of manual errors. The intention is that expression overriding is used sparingly and carefully; a large number of overriding expressions is undesirable, difficult to manage and defeats some of the purpose of the inheritance mechanism. The issue is that there is no mechanism (e.g. syntax checking) at present to stop the analyst from overriding every behaviour from an inherited pattern within a specialisation, undermining the well-known behaviour. At the same time, the method should remain as flexible as possible, and so this is a feature which may need improvement. Ultimately, as with other safety analysis methods, a certain level of expertise is assumed on behalf of the analyst to ensure the method is used correctly.

5.2 Further work

The aim of improving reuse within safety analysis has been achieved to some extent, but is far from solved. With regards to the solution presented in this thesis, though some improvement to reusability has been made, at the same time it has raised questions about the additional effort introduced. One contentious point is whether these extra requirements are worth the reduction in annotation effort. The validation study has given positive results, but additional research would be very beneficial.

A definitive answer about the effectiveness of patterns of failure behaviour might only be gained through several full-scale analyses performed with the GFL method and comparing levels of achievable reuse with an identical analysis using the legacy annotation method (though it may be possible even then the result may be system or analysis specific, thus the need for several studies). It would be reasonable to expect to see in such a study that many common patterns of behaviour would be found, providing an opportunity to quantify the extent of reuse. Indeed as more general behaviours are discovered and more patterns created (expanding the library of available patterns), the method will be increasingly beneficial. Ultimately it was deemed impractical to perform such a large scale study in the course of this thesis, having already gained positive results from the smaller case studies.

If a wider study can confirm the efficacy of the new method, it would be reasonable to extend the language further with new operators, enabling the capture of other generalisations of potentially more complex behaviour. This was briefly investigated with the use of a 'majority voting' operator to demonstrate this ability, and so could be expanded to include other common voting behavioural types, such as minority combinations.

Recent developments in HiP-HOPS for temporal logic suggest that temporal operators could also be included in an extended generalised language. Based on work in the PANDORA project by Walker (2009), the definitions given to priority-logic could be incorporated into the GFL, for instance with operators that generalise the sequencing of events, like 'before' and 'after'. The possibility is that *dynamic patterns of behaviour*, where the sequence of events is critical to the component's failure behaviour, could be captured, stored and reused.

Finally, the important issues of dependence of safety analysis on state and application context and how it affects the use and reuse of GFL specifications are areas that need to be further investigated.

A small step in extending a state-of-the-art technique for safety analysis towards enabling more generalisable and potentially reusable component failure specifications has been achieved. Many more steps are required to realise some of the potential that this thesis hopes to have shown.

References

Andrews J D, 1998, Fault Tree Analysis Tutorial, *Proceedings of the 16th International System Safety Conference*, Seattle, Washington, 1998.

ARP 926, 1967, Design Analysis Procedure For Failure Modes, Effects and Criticality Analysis (FMECA), *Society for Automotive Engineers*, ARP926.

Avižienis A, Laprie J-C, Randell B, 2001, Fundamental Concepts of Dependability, Research Report N01145, LAAS-CNRS.

Bozzano M and Villafiorita A, 2003, Improving system reliability via model checking: The FSAP/NuSMV-SA safety analysis platform, *International Conference on Computer Safety, Reliability, and Security, SAFECOMP 2003* (*Lecture Notes in Computer Science*, vol. 2788), pp 49-62.

Cardelli L and Wegner P, 1985, On Understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys*, **17**(4), pp 471-522.

Chen D-J, Johansson R, Lönn H, Papadopoulos Y, Sandberg A, Törner F, Törngren M, 2008, Modelling support for design of safety-critical automotive embedded systems, *Computer Safety, Reliability, and Security, 27th International Conference, SAFECOMP 2008* (*Lecture Notes in Computer Science*, vol. 5219), pp 72-85.

Ciardo G and Lindermann C, 1993, Analysis of deterministic and stochastic Petri nets, In *Proceedings of the 5th International Workshop on Petri nets and Performance models* (*PNPM 1993*).

Clarke E M, Grumberg O, Peled D A, 1999, *Model Checking*, MIT Press.

Dehlinger J and Lutz R, 2006, PLFaultCAT: A Product-Line Software Fault Tree Analysis Tool, *Automated Software Engineering*, **13**(1), pp 169-193.

Ezhilchelvan P and Shrivastava S, 1989, A Classification of Faults in Systems, University of Newcastle upon Tyne.

Feiler P H and Rugina A-E, 2007, Dependability Modeling with the Architecture Analysis and Design Language (AADL), *Technical Report*, *CMU/SEI-2007-TN-043*.

Feiler P H, Gluch D P, Hudak J J, 2006, The Architecture Analysis and Design Language (AADL): An Introduction, *Technical Report*, *CMU/SEI-2006-TN-011*.

Fenelon P, McDermid J A, Nicolson M, Pumfrey D J, 1994, Towards integrated safety analysis and design, *ACM SIGAPP Applied Computing Review*, **2**(1), pp 21-32.

Fenelon P and McDermid J A, 1993, An integrated toolset for software safety analysis, *Journal of Systems and Software*, **21**(3), pp 279-290.

Ferguson S A, 2007, The effectiveness of electronic stability control in reducing real-world crashes: A literature review, *Traffic Injury Prevention*, **8**(4), pp 329-338.

Gamma E, Helm R, Johnson R, Vlissides J, 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Ge X, Paige R F, McDermid J A, 2009, Probablistic Failure Propagation and Transformation Analysis, *Computer Safety, Reliability, and Security*, **5775**, pp 215-228.

German R, and Mitzlaff J, 1995, Transient analysis of deterministic and stochastic Petri nets with TimeNET, *Proceedings of the 8th International Conference on Computer Performance Evaluation, Modelling Techniques, and Tools and MMB* (*Lecture Notes in Computer Science*, vol. 977), pp 209-223.

Gould J, Glossop M, Ioannides A, 2000, Review of hazard identification techniques, *Health and Safety Executive*, HSL/2005/58.

Grunske L, Kaiser B, Papadopoulos Y, 2005, Model-driven Safety Evaluation with State-event based Component Failure Annotations, *8th International Symposium on Component-based Software Engineering*, pp 33-48.

Grunske L and Neumann R, 2002, Quality improvement by integrating non-functional properties in software architecture specification, *EASY'02: Second Workshop on Evaluating and Architecting System Dependability*, pp 23-32.

Hamann R, Uhlig A, Papadopoulos Y, Rüde E, Grätz U, Lien R, 2008, Derivation or ship system safety criteria by means of risk-based ship system safety analysis, *27th*

International Conference on Offshore Mechanics and Arctic Engineering (*OMAE'08*).

Havelund K, Lowry M, Penix J, 2001, Formal Analysis of a Space Craft Controller using SPIN, *IEEE Transactions on Software Engineering*, **27**(8).

Heilmann R, Rothbauer S, Sutor A, 2007, Component fault tree analysis resolves complexity: Dependability confirmation for a railway brake system, *SAFECOMP 2007* (*Lecture Notes in Computer Science*, vol. 4680), pp 100-105.

Heiner G, Thurner T, 1998, Time-triggered architecture for safety-related distributed real-time systems in transportation systems, *Proceedings of FTCS-28*, pp 402-407.

IEC 60812, 2006, Analysis techniques for system reliability - Procedure for failure mode and effects analysis (FMEA).

IEC 61508, 1997, Functional safety of electrical/electronic/programmable electronic safety-related systems.IEC 61025, 1990, Fault-Tree-Analysis (FTA).

Johnson R, 1997, Components, Frameworks, Patterns, *Communications of the ACM*, vol. 40, pp 10-17.

Joshi A, Vestal S, Binns P, 2007, Automatic Generation of Static Fault Trees from AADL Models, *DSN Workshop on Architecting Dependable Systems*, Edinburgh 2007.

Kaiser B, Gramlich C, Forster M, 2007, State/event fault trees - A safety analysis model for software-controlled systems, *Reliability Engineering and System Safety*, **92**(11), pp 1521-1537.

Kaiser B, Gramlich C, 2004, State-Event-Fault-Trees - A Safety Analysis Model for Software Controlled Systems, Computer, *SAFECOMP 2004* (*Lecture Notes in Computer Science*, vol. 3219), pp 195-209.

Kaiser B, Liggesmeyer P, Mäckel O, 2003, A new component concept for fault trees, *Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software* (*SCS'03*), pp 37–46.

Karunanithi S, Bieman J M, 1993, Measuring software reuse in object oriented systems and ada software, *Technical report CS-93-125*, Department of Computer Science, Colorado State University.

Kehren C, Seguin C, Bieber P, Castel C, Bougnol C, Heckmann J-P, Metge S, 2004, Architecture patterns for safe design, *Proceedings of the first AAAF Conference on Complex and Safe System Engineering*.

Kelly T P and Weaver R A, 2004, The Goal Structuring Notation - A Safety Argument Notation, *Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases*.

Kelly T P, 2001, Concepts and Principles of Compositional Safety Case Construction, *Research Report COMSA/2001/1/1*.

Kelly T P, McDermid J A, 2001, A Systematic Approach to Safety Case Maintenance, *Reliability Engineering and System Safety*, vol. 71, pp 271-284.

Kelly T P, 1998, Arguing Safety - A Systematic Approach to Managing Safety Cases, PhD dissertation, University of York.

Kelly T P and McDermid J A, 1998, Safety Case Patterns - Reusing Successful arguments, In *Proceedings of IEE Colloquium on Understanding Patterns and Their Application to System Engineering*.

Kelly T P and McDermid J A, 1997, Safety Case Construction and Reuse using Patterns, In *Proceedings of 16th International Conference on Computer Safety, Reliability and Security* (*SAFECOMP 1997*).

Kletz T A, 1997, HAZOP – Past and Future, *Reliability Engineering and System Safety*, **55**(3), pp 263-266.

Kopetz H, Grünsteidl G, 1994, TTP – a protocol for fault-tolerant real-time systems, *IEEE Computer*, **27**(1), pp 14-23.

Kopetz H, Damm A, Koza C, Mulazzani M, Schwabl W, Senft C, Zainlinger R, 1989, Distributed fault tolerant real-time systems: the MARS approach, *IEEE Micro*, **9**(1), pp 25-40.

Leveson N G, 1995, *Safeware: System Safety and Computers*, Addison-Wesley.

Lisagor O, McDermid J A, Pumfrey D J, 2006, Towards a practicable process for automated safety analysis, *Proceedings of ISSC 2006*.

Lutz R, Helmer G, Moseman M, Statezni D, Tockey S, 1998, Safety Analysis of Requirements for a Product Family, *Proceedings of the 3rd International Conference on Requirements Engineering (ICRE'98)*.

Mäckel O, Rothfelder M, 2001, Challenges and Solutions for Fault Tree Analysis Arising from Automatic Fault Tree Generation: Some Milestones on the Way, *ISAS-SCI* (1), pp 583-588.

Paige R F, Rose L M, Ge X, Kolovos D S, Brooke P J, 2009, FPTC: Automated Safety Analysis for Domain-Specific Languages, *Lecture Notes in Computer Science* vol. 5421, pp 229-242.

Papadopoulos Y, Walker M, Reiser M-O, Weber M, Servat D, Abele A, Johansson R, Lonn H, Torngren M, Sanberg A, 2010, Automatic Allocation of Safety Integrity Levels, *8th European Dependable Computing Conference – CARS workshop*, Valencia, Spain, April 2010.

Papadopoulos Y and Grante C, 2005, Evolving car designs using model-based automated safety analysis and optimisation techniques, *Journal of Systems and Software*, **76**(1), pp 77-89.

Papadopoulos Y, Parker D, Grante C, 2004, Automating the failure modes and effects analysis of safety critical systems, *International Symposium on High-Assurance Systems Engineering* (*HASE 2004*), pp 310-311.

Papadopoulos Y, Petersen U, 2003, Combining ship machinery system design and first principle safety analysis, *8th International Marine Design Conference* (*IMDC 2003*), Athens, pp 415-426.

Papadopoulos Y, Maruhn M, 2001, Model-based synthesis of fault trees from matlab-simulink models, *International Conference on Dependable Systems and Networks* (*DSN 2001*), pp 77-82.

Papadopoulos Y, McDermid J A, Sasse R, Heiner G, 2001, Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure, *Reliability Engineering and System Safety*, **71**(3), pp 229-247.

Papadopoulos Y and McDermid J A, 1999, Hierarchically performed hazard origin and propagation studies, In *18th International Conference in Computer Safety, Reliability and Security*, Toulouse, France, pp 139-152.

Parker D J, Walker M D, Papadopoulos Y, Grante C, 2006, Component-Based, Automated FMEA of Advanced Active Safety Systems, In *31st World Automotive Congress FISITA*, Yokohama, Japan.

Paynter S E, Armstrong J A, Haveman J, 2000, ADL: An activity description language for real-time networks, *Formal Aspects of Computing*, **12**(2), pp 120-140.

Price C J and Taylor N S, 2002, Automated Multiple Failure FMEA, *Reliability Engineering and System Safety*, **76**(1), pp 1-10.

Rauzy A, 2002, Modes automata and their compilation into fault trees, *Reliability Engineering and System Safety*, **78**(1), pp 1-12.

Redmill F, Chudleigh M F, Catmur J R, 1997, Principles underlying a guideline for applying HAZOP to programmable electronic systems, *Reliability Eningeering and System Safety*, **55**(3), pp 283-293.

Rugina A-E, Kanoun K, Kaâniche M, 2007, A System Dependability Modeling Framework Using AADL and GSPNs, *Lecture Notes in Computer Science* vol. 4612, pp 14-38.

Rugina A-E, 2005, System Dependability Evaluation using AADL (Architecture Analysis and Design Language), *Rencontres Jeunes Chercheurs en Informatique Temps Réel* (*RJCITR*).

Smith S P and Harrison M D, 2005, Measuring reuse in hazard analysis, *Reliability engineering and system safety*, **89**(1), pp 93-104.

Smith S P, Harrison M D, 2002, Improving hazard classification through the reuse of descriptive arguments, *Lecture Notes in Computer Science* vol. 2319, pp 255-268.

Thums A, Schellhorn G, 2003, Model Checking FTA, *Lecture Notes in Computer Science* vol. 2805, pp 739-757.

Mathworks, 2009, MATLAB software, www.mathworks.com [Accessed 5th May 2010].

Uhlig A, Kurzbach G, Hamann R, Papadopoulos Y, Walker M, Lühmann B, 2007, Simulation model based risk and reliability analysis, *Tagung Technische Zuverlässigkeit*, *Annual VDI reliability conference*, Stuttgart, April 2007.

Vesely W E, Stamatelatos M, Dugan J, Fragola J, Minarick J, Railsback J, 2002., *Fault Tree Handbook with Aerospace Applications*, NASA office of Safety and Mission Assurance.

Vesely W E, Goldberg F F, Roberts N H, Haasl D F, 1981, *Fault Tree Handbook*. Washington D.C., U.S. Nuclear Regulatory Commission.

Walker M, 2009, Pandora: A Logic for the Qualitative Analysis of Temporal Fault Trees, PhD dissertation, University of Hull.

Wallace M, 2005, Modular architectural representation and analysis of fault propagation, *Electronic Notes in Theoretical Computer Science*, **141**(3), pp 53-71.

Wilson S, Kelly T P, McDermid J A, 1997, Safety Case Development: Current Practice, Future Prospects, *Safety and Reliability of Software Based Systems -Twelfth Annual CSR Workshop*, Bruges, Belgium, 1997.

Wilson S, McDermid J A, Fenelon P, Kirkham P, 1995, No More Spineless Safety Cases: A Structured Method and Comprehensive Tool Support for the Production of Safety Cases, *Institution of Nuclear Engineers Conference '95 2nd International Conference on Control and Instrumentation in Nuclear Installations*, April 1995.

Wolforth I, Walker M, Grunske L, Papadopoulos Y, 2010a, Generalisable Safety Annotations for Specification of Failure Patterns, *Software: Practice and Experience*, **40**(5), pp 453-483.

Wolforth I, Walker M, Papadopoulos Y, Grunske L, 2010b, Capture and Reuse of Composable Failure Patterns, *International Journal of Critical Computer-Based Systems*, **1** (1-3), pp 128-147.

Wolforth I, Walker M, Papadopoulos Y, 2008, A language for failure patterns and application in safety analysis, *IEEE Dependable Computing Systems* (*DEPCOS 2008*), pp 47-54, June 26-28 2008, Szklarska Poręba, Poland.

Wolforth I, 2005, Combining fault tree synthesis with efficient evaluation of fault trees, MSc dissertation, University of Hull.