Using Safety Contracts in the Development of Safety Critical Object-Oriented Systems

Richard D. Hawkins

This thesis is submitted in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy.

University of York
York
YO10 5DD
UK

Department of Computer Science

March 2006

For my Mum and Dad

and

Heather

Abstract

Developers of safety critical software are becoming increasingly interested in using the object-oriented paradigm. If a developer is to use an object-oriented approach successfully in safety critical applications they must be able to demonstrate that the resulting software system is sufficiently safe to operate. There are a number of existing approaches to developing safe software however these cannot be used effectively for an object-oriented approach.

This thesis identifies the inadequacies of the existing approaches to reasoning about the safety of safety critical object-oriented systems. It identifies a need for a systematic, thorough and scalable process to identify the properties required of software objects to adequately address their contribution to system-level hazards. This thesis describes and evaluates such an approach which addresses this need. The proposed approach uses the concept of safety contracts to define safety requirements between objects in the system design. An analysis process is described which can be used to generate these contracts. It is shown how the safety contracts can be used in constructing a safety argument which demonstrates that the system is acceptably safe.

Throughout the thesis, how the approach is applied is illustrated with realistic examples. A case study, taken from an industrial application, is also provided and used in the evaluation.

Contents

Abstract **3**

Acknowledgement **13**

Declaration **14**

1 Introduction **15**

1.1 The use of computers in safety critical systems 15

1.1.1 Ariane 5 15

1.1.2 Increasing complexity 16

1.2 The object-oriented paradigm 18

1.3 Using safety contracts to ensure the safety of systems developed using an OO approach 19

1.4 Thesis proposition 19

1.5 Thesis structure 20

2 Survey of Related Literature **22**

2.1 Introduction 22

2.2 The development of safety-critical software systems 22

2.2.1 Safety standards 23

2.2.2 Safety critical system development 29

2.2.3 The software safety case 45

2.3 The object oriented paradigm 49

2.3.1 Object oriented concepts . . . 49
2.3.2 Object oriented analysis and design . . . 50
2.3.3 Software contracts . . . 58
2.3.4 Analysing OO designs . . . 63
2.3.5 Representing requirements . . . 68
2.3.6 Verification of OO . . . 70
2.4 Thesis Contribution . . . 76
2.5 Conclusions . . . 77

3 Performing Safety Analysis of OO Systems 79

3.1 Introduction . . . 79
3.2 A Framework for Analysis . . . 80
3.3 Developing an analysis process . . . 83
3.4 An Aircraft Stores Management System . . . 84
3.5 Analysing Functional Aspects . . . 87
3.5.1 Step One: Identify Hazards . . . 87
3.5.2 Step Two: Define hazardous software failure modes . . . 88
3.5.3 Step Three: Identify basic failure events . . . 89
3.5.4 Step Four: Investigate causes of failures . . . 91
3.5.5 Step Five: Define hazardous object behaviour . . . 97
3.6 Analysing Temporal Aspects . . . 99
3.6.1 Step One: Split scenario into tasks . . . 99
3.6.2 Step Two: Investigate effects of timing deviations . . . 100
3.6.3 Step Three: Analyse alternative scenarios . . . 102
3.6.4 Step Four: Define timing requirements . . . 104
3.7 Analysing Value Aspects . . . 107
3.7.1 Step One: Identify critical data . . . 107
3.7.2 Step Two: Identify manipulators . . . 108
3.7.3 Step Three: Define constraints for critical data . . . 109

3.8 Conclusions . . . 110

4 Safety Contracts 111

4.1 Introduction . . . 111

4.2 The Need for Contracts . . . 112

4.3 Defining Safety Contracts . . . 115

4.3.1 Notation for Safety Contracts . . . 117

4.3.2 Safety Contracts for the SMS . . . 119

4.4 Utilisation of Safety Contracts . . . 121

4.4.1 Supporting Design Change through Safety Contracts . . . 124

4.4.2 Supporting the Reuse of Design Elements through Safety Contracts . . . 132

4.5 Conclusions . . . 133

5 Creating a Safety Argument for OO Systems 134

5.1 Introduction . . . 134

5.2 Modular Safety Argument Structures . . . 135

5.3 Developing the Safety Arguments . . . 138

5.3.1 Software System Level Argument . . . 139

5.3.2 Interactions Argument . . . 140

5.3.3 Class Argument . . . 140

5.4 Handling Change and Reuse . . . 141

5.4.1 Changes to the Design of a Class . . . 141

5.4.2 Introducing a New Class . . . 142

5.4.3 Reusing a Class . . . 143

5.5 Conclusions . . . 144

6 Aircraft Avionics Control System; A Case Study 146

6.1 Introduction . . . 146

6.2 System Overview . . . 146

6.3 Safety Analysis of ACS . . . 147

6.3.1 Functional Analysis 147
6.3.2 Temporal Analysis 153
6.3.3 Value Analysis 155
6.4 Defining Safety Contracts for the System 156
6.4.1 Identify Safety Obligations 157
6.5 Creating a Safety Argument for the ACS 158
6.6 Conclusions 158

7 Evaluation 160

7.1 Introduction 160
7.2 Systematic Approach 161
7.3 Thorough Approach 163
7.4 Scalability 164
7.5 Evaluation Against Problem Statements 166
7.6 Conclusions 167

8 Conclusions 169

8.1 Concluding Remarks 169
8.1.1 Conclusions on the analysis process contribution 169
8.1.2 Conclusions on the use of safety contracts 170
8.1.3 Conclusions on the safety argument patterns 170
8.2 Further work areas 171
8.2.1 Verification Evidence 171
8.2.2 Safety Contract Enforcement 171
8.2.3 System Implementation 171
8.3 Overall conclusions 172

A Software System Argument Module Pattern 173

B Interactions Argument Module Pattern 177

C Class Argument Module Pattern **181**

D ACS Case Study Reference Material **185**

List of Figures

1.1 Growth of airborne software: Military 17

2.1 Verification of outputs of software coding and integration processes from standard DO178-B 25

2.2 V lifecycle model showing safety activities 30

2.3 The decomposition of a platform design 31

2.4 Risk assessment matrix 32

2.5 Main elements of the fault tree notation 34

2.6 A simple fault tree 35

2.7 HAZOP guidewords 36

2.8 Example attribute guide word interpretations for data flow diagrams 37

2.9 Example attribute guide word interpretations for state transition diagrams . . . 38

2.10 Failure classifications used to structure SHARD guidewords 38

2.11 Example SHARD guidewords for MASCOT 3 39

2.12 An FMEA table 42

2.13 Representation of the safety critical systems development process 45

2.14 Main elements of the GSN notation 46

2.15 An example GSN goal structure 46

2.16 GSN Argument Pattern for Component Contributions to System Hazards 47

2.17 GSN Argument Pattern for Hazardous Software Failure Mode Decomposition . . 48

2.18 GSN Argument Pattern for Hazardous Software Failure Mode Classification . . . 48

2.19 UML class diagram 52
2.20 UML sequence diagram 53
2.21 UML collaboration diagram 53
2.22 UML state chart diagram 54
2.23 UML use case diagram 55
2.24 An example OCL constraint 58
2.25 Redefinition of a routine under contract 61
2.26 Splitting a Class into Slices 73
2.27 Matrix structure for testing derived classes 74

3.1 Deriving safety requirements for a software system 80
3.2 Identifying software contributions for functional (left) and OO (right) systems . . 82
3.3 Sequence of interactions occurring in an OO system 82
3.4 UML class diagram for aircraft SMS 85
3.5 UML sequence diagram for release of store 86
3.6 Overview of safety analysis for functional behaviours of objects 87
3.7 Fault tree for SMS HSFM 90
3.8 SHARD guide word interpretation used for operation calls 92
3.9 SHARD analysis of checkWOW() interaction 93
3.10 UML statechart for the SMS store object 95
3.11 Mutated statechart for SMS store object 96
3.12 Overview of safety analysis for temporal behaviour of objects 99
3.13 Timing deviations applied to tasks for the SMS 102
3.14 Possible alternative scenarios for release of a store 104
3.15 Timing constraints on tasks in a scenario 105
3.16 Sequence diagram representing a task 106
3.17 Safety analysis process for value aspects 107
3.18 Manipulators identified for critical data in the SMS 109

4.1 Identifying safety obligations for an object 113
4.2 Safety obligations identified for the store class 123
4.3 Changed UML sequence diagram for release of store 127
4.4 SHARD analysis of deleteStore() 128
4.5 Using inheritance to create new classes 129
4.6 Operation redefinition using inheritance 130

5.1 Monolithic safety argument structure 135
5.2 Modular argument structure including class modules 137
5.3 Modular argument structure with separate interactions argument 138

D.1 System architecture diagram for the ACS 185
D.2 Use case diagram: Maintain minimum height 186
D.3 Sequence diagram: Maintain minimum height - TF enabled 187
D.4 Fault tree for ACS fails to warn pilot when aircraft altitude falls below minimum safe altitude 188
D.5 SHARD analysis of interaction 1 188
D.6 SHARD analysis of interaction 2 189
D.7 SHARD analysis of interaction 3 189
D.8 SHARD analysis of interaction 4 189
D.9 SHARD analysis of interaction 5 190
D.10 SHARD analysis of interaction 6 190
D.11 Statechart model for the Navigator class 191
D.12 Mutated statechart for the Navigator class 192
D.13 Applying timing deviations to tasks 193
D.14 Alternative scenarios 193
D.15 Manipulators of critical data 194
D.16 Safety obligations for the Flight Director class 194
D.17 Safety obligations for the Flight Director class 194

D.18 Safety obligations for the Navigator class 195

D.19 Safety obligations for the INS class 195

D.20 Safety obligations for the Control Panel class 196

D.21 Safety obligations for the Aircraft class 196

D.22 Software system level argument for the ACS 197

D.23 Interactions argument for the ACS 198

D.24 Navigator class argument for the ACS 199

Acknowledgement

I would like to thank my supervisor John McDermid whose wisdom and guidance have been invaluable to me. I am indebted to John for his patience and encouragement over the last four years.

I would also like to thank everyone from BAE Systems and MBDA who I have worked with through the DCSC, for their support. In particular I would like to thank Brian Jepson and Jane Fenn for all their assistance.

I would like to thank my colleagues and friends at York who I have been honoured to work with. In particular I would like to thank Rob Weaver, Frantz Iwu, Rob Collyer, Paul Emberson, Rob Alexander, Martin Hall-May, Simon Bates, Kester Clegg, Philippa Conmy, Tim Kelly, Iain Bate, David Pumfrey and Mark Nicholson.

I would like to thank my family, particularly my Mum and Dad, whose love and encouragement are a huge inspiration to me.

Final thanks go to my wife, Heather, who keeps me happy and sane, and whom I adore.

Declaration

Some of the material presented in this thesis has previously been published in the following papers:

- Richard Hawkins, Simon Bates and John McDermid, "Developing Successful Modular Arguments for Object Oriented Systems", 22nd ISSC, Providence, RI 2004
- Richard Hawkins, Ian Toyn and Iain Bate, "An Approach to Designing Safety Critical Systems using the Unified Modelling Language ", Critical Systems Development with UML, UML '03 workshop, San Fransisco 2003
- Iain Bate, Richard Hawkins and John McDermid, "A Contract-based Approach to Designing Safe Systems", 8th Australian Workshop on Safety Critical Systems and Software (SCS'03), Canberra 2003
- R. D. Hawkins, J. A. McDermid and I. J. Bate, "Developing Safety Contracts for OO Systems", 21st ISSC, Ottawa 2003
- R. D. Hawkins and J. A. McDermid, "Performing Hazard and Safety Analysis of OO Systems", 20th ISSC, Denver 2002

Except where stated, all of the work contained within this thesis represents the original contribution of the author.

Chapter 1

Introduction

1.1 The use of computers in safety critical systems

Safety is freedom from accidents. System safety is concerned with preventing foreseeable accidents, and minimising the results of unforeseen ones [40]. There are many examples of systems whose failure could lead to accidents. Such systems are most often found within the aerospace, rail, automotive and medical sectors as well as in industries such as manufacturing and nuclear. Such systems have been in existence in one form or another since the industrial revolution. As such, a body of knowledge and expertise has grown up around their development to reduce the risk of accidents. A more recent development has been the use of computers as part of these systems. Computers are integrated into the operation of control systems to enhance performance or increase efficiency. Such computer systems are often referred to as safety-critical systems, as their failure may lead to an accident. The software that executes on these computers plays a crucial role in ensuring the safe operation of the system. Errors in the software, which may appear at first sight to be minor, or even trivial, can result in catastrophic consequences. This is illustrated quite clearly using the example of the Ariane 5 accident.

1.1.1 Ariane 5

On 4th June 1996, the maiden flight of the European Ariane 5 rocket broke up and exploded about 40 seconds after take off. Although there were fortunately no crew on board, the financial loss was reported to be half a billion dollars. The international inquiry board concluded that the explosion was the result of a software error. The error arose in ten year old software which had

been reused from the Ariane 4 rocket. The software related to the Inertial Reference System, which performs computations before lift-off. These computations are not required after lift-off, but continue anyway due to the time taken to reset. The computation involves the conversion of the horizontal bias from 64-bit to 16-bit. After lift-off Ariane 5 produced a horizontal bias that was too large for 16-bits. This raised an exception which wasn't caught and handled, and thus led to catastrophe. On Ariane 4, it had been determined that such an error could not occur, so no error handling mechanism had been necessary. As Ariane 5 had a different launch trajectory to Ariane 4, the error was, as proved to be the case, now possible. The designers of Ariane 5 had not sufficiently checked the reused software to ensure it remained safe in the new system. This illustrates two key points. Firstly that small errors in software can lead to very large (and costly) outcomes. Secondly the importance of capturing all requirements which could impinge upon the safety of the system, and how this is particularly important where software may be reused in a different system at some future time. Further details on this accident can be found in the accident report [41].

1.1.2 Increasing complexity

Not only is software becoming more prevalent in safety critical systems, the software that is being used is also becoming increasingly more complex. Figure 1.1 [80] illustrates how the number of lines of code (LOC) used in software on military aircraft has increased over the last couple of decades.

This can cause many problems for safety critical systems developers. In order to certify a software system it is necessary to demonstrate that the software will not contribute to an accident. This can be very difficult even for very trivial amounts of computer code. An example taken from [80] illustrates the impossibility of attempting to completely test even tiny amounts of software code. If we consider trying to exhaustively test 4 data items, each of which is of 16 bits in size. Each of the bits has two states (0 or 1), so the total number of permutations for the four data items is 2^{64} (or approximately 10^{19} permutations). If one test were performed every millisecond, it would take around 500,000 years to test every combination. Clearly it is not going to be possible to use this as a method of gaining confidence in the safety of large computer systems.

Due to the difficulty in demonstrating the acceptability of software, the tendency within the safety critical industry has been, perhaps reassuringly, one of caution, or at least of “sticking with what you know”. This means that many of the new technologies and paradigms which

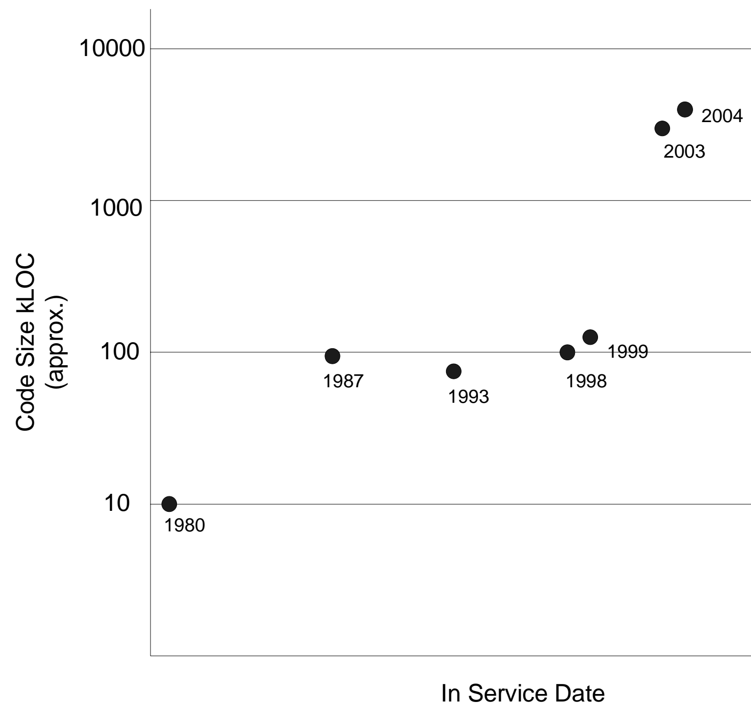


Figure 1.1: Growth of airborne software: Military

have emerged and been used in other software engineering domains have not been used in safety critical systems, or have been adopted slowly. There are two main reasons for this. The first reason is that for most safety critical applications, the increase in computing power that the developments bring is unnecessary. Therefore the corresponding technologies, and inherent complexities, are also not required. The second reason is that a large amount of confidence in the software, and therefore support for its safety, can be gained from experience of using trusted technologies and techniques, and providing evidence of historical safe operation. Without this previous service history and experience, this crucial evidence is unavailable.

As the demands made on the software in safety critical systems has continued to increase, for example with the move towards fly-by-wire aircraft, the complexity of the software has also continued to increase. The result of this is that many of the advances in software engineering, such as technologies and paradigms previously not required in safety critical applications, are now having to be considered. This presents challenges to developers of safety critical systems, who must find ways of certifying software developed employing these advancements.

1.2 The object-oriented paradigm

One such advancement has been the development of the object-oriented (OO) paradigm. It should be noted that OO could in no way be considered a particularly *new* paradigm. The basic concepts of OO first emerged with the development of the Simula programming languages [63] in the 1960's. The concepts of OO will be discussed in chapter 2, however OO was developed as a way of implementing large-scale complex programs through the use of a large number of interacting components.

OO has become increasingly popular in non-safety critical fields. There are many potential reasons for this popularity, not least of which is the prevalence of OO programming languages such as C++ and Java. Anecdotally it is easy to find claims and counter claims about the benefits and drawbacks of OO. This thesis does not aim to contribute to this debate, or indeed to promote the use of an OO approach. However, the increasing popularity (indeed, near ubiquity) in other domains suggests that there must be benefits to developing in such a manner that are worth embracing, if feasible, in the safety critical domain.

The benefits that have been claimed for an OO approach include increased changeability of the software due to abstraction and encapsulation, and increased facilities for reuse of software elements through generalisation and inheritance. This shall be discussed in more detail in chapter 2. As well as these benefits, in a drive to minimise the effects of hardware obsolescence, particularly within the defence systems arena, there is a move towards the use of middleware such as the Common Object Request Broker Architecture (CORBA), and portable languages such as java. These technologies are object oriented.

The challenge facing developers wishing to make use of OO technologies in safety critical software is gaining confidence that the software is acceptably safe[1]. A number of questions are raised when considering this, such as whether a traditional safety approach can be used successfully to demonstrate the safety of a system developed using an OO approach. If this is not the case, then what new approach is needed such that safety can be assured? This thesis will investigate the shortcomings of existing approaches when applied to OO systems, before proposing an alternative approach which enables the safety of an OO system to be demonstrated.

[1]Strictly, software itself cannot be safe or unsafe. It is only within the context of the entire system that the safety of software can be judged.

1.3 Using safety contracts to ensure the safety of systems developed using an OO approach

It is crucial to the safe use of any system which uses software that safety requirements can be derived which constrain the design of the software such that it will not contribute to accidents. This is equally true for systems using an OO approach. In this thesis it is proposed that safety contracts be used to capture the safety requirements placed on the OO software design. It is shown how this approach enables the safety of an OO system to be demonstrated. A safety contract approach ensures that there is traceability between the evidence provided in the safety argument (see section 2.2.3), the software system design and the system hazards which must be controlled. As shall be seen, establishing such traceability is one of the key challenges for demonstrating the safety of OO systems.

It is crucial to the success of any safety process that it is integrated with the existing system and software development processes. A safe system can only be achieved if safety is considered from the earliest stages of the system's development, right through to its operation and maintenance. The safety activities must therefore be seen as part of the development process for the software, rather than a separate activity. It shall be shown that the process proposed in this thesis can be integrated with existing system and software development processes used for developing OO software.

In order to be effective, the approach developed to ensure the safety of an OO system must not unduly impinge upon the utility of that system. As stated earlier, there are certain advantages which developers perceive to get through the adoption of an OO approach such as a robustness to change, and a greater potential for reuse. This thesis will show how a safety contract approach helps to preserve these beneficial properties whilst ensuring the system remains safe.

1.4 Thesis proposition

The proposition adopted in this thesis can be stated as follows:

> Through the development of safety contracts, it is possible to establish a systematic, thorough and scalable process to identify the properties required of software objects to adequately address their contribution to system-level hazards.

1.5 Thesis structure

The thesis is divided into the following chapters:

Chapter 2 presents a survey of the published literature in the areas of software safety and the OO paradigm. The survey first looks at the current safety processes used for software systems and discusses the analysis techniques that are used. This includes a review of the use of safety arguments as part of the safety process. The OO paradigm is discussed, along with the use of software contracts in designing OO systems. The survey then goes on to focus more specifically on literature that deals with the safety of OO systems. The survey will assess the applicability of conventional software safety approaches to software developed using an OO approach. The limitations of current approaches specifically used in analysing OO systems are also assessed.

Chapter 3 proposes a process for analysing OO systems in order to identify potentially hazardous behaviour. The analysis focusses on the interactions which occur between objects in the system and applies existing techniques to identify failures in the interactions that could contribute to hazards in the system.

Chapter 4 shows how the output generated by the analysis in Chapter 3 can be used to generate a set of derived safety requirements (DSRs) on the interactions. The DSRs are specified in the form of safety contracts between objects. It is shown how these safety contracts can be used to identify safety obligations upon the objects in the system.

Chapter 5 develops safety argument patterns which can be used to create an argument concerning the safety of an OO system developed using the process described in the thesis. A modular structure is used for the safety argument. It is shown how this structure increases the ability to safely deal with changes to the system design, and the reuse of design artifacts.

Chapter 6 describes a case study which demonstrates how the process developed in the thesis, can be applied to a realistic system. A safety argument is produced based upon the analysis performed, and the patterns presented in Chapter 5. The case study helps to illustrate the effectiveness of the approach.

Chapter 7 describes how the process developed in chapters 3, 4 and 5 has been evaluated. It describes the extent to which the work presented in the previous chapters satisfies the thesis proposition. The evaluation is based upon the case study in chapter 6, as well as

assessment against a set of defined criteria and problem statements identified as part of the literature survey.

Chapter 8 presents the conclusions that are drawn from the thesis. Potential areas of future research are also identified.

Chapter 2

Survey of Related Literature

2.1 Introduction

This chapter contains a survey of the published literature related to the proposition of this thesis. The survey is split into two main sections. The first section reviews current literature relating to the development and analysis of software for safety critical systems. This identifies current approaches for developing safety critical software, and reviews the methods and analysis techniques available to system developers. The concept of a safety case is explored, and literature regarding the development of safety arguments is reviewed.

The second section firstly discusses the basic concepts of the OO paradigm. Approaches to object oriented development are explored as well as notations used in designing OO systems. This will include methodologies used for developing real-time systems. The use of software design contracts, as part of an OO development process is also considered. Finally, literature relating to the use of OO for safety-critical systems is reviewed and evaluated. At the time of commencing work on this thesis, the amount of literature directly dealing with OO software in safety critical systems was limited.

2.2 The development of safety-critical software systems

Safety critical software is any software that can directly or indirectly contribute to the occurrence of a hazardous system state [40]. A hazard can be defined as a physical situation, often following from some initiating event, that can lead to an accident [49]. When considering safety critical software, the aim is to prevent accidents by ensuring the software does not contribute to any

hazards. This section reviews current practice for ensuring the safety of software systems.

2.2.1 Safety standards

It is perhaps best to start by considering the various standards that exist for industries which develop safety critical systems. The standards are intended to represent collective wisdom, or best practice within the industry to which they apply. This survey will focus on three important standards used in different industry sectors. Although the detailed requirements in the standards vary between industry sectors, and individual standards, this survey aims to identify the similarities between the approaches, whilst noting any important variations. This identifies the framework within which most safety critical systems are developed.

2.2.1.1 RTCA/DO-178B

The main standard used to certify software used in aircraft in the US and Europe is RTCA/DO-178B [66]. The most important concepts in this standard are 'failure condition categorisation' and 'software level definitions'. Failure conditions are categorised in one the five categories defined below:

catastrophic failure conditions which prevent continued safe flight and landing.

hazardous / severe-major failure conditions which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be:

1. a large reduction in safety margins or functional capabilities
2. physical distress or higher workload such that the flight crew could not be relied on to perform their tasks accurately or completely
3. adverse effects on occupants including serious or potentially fatal injuries to a small number of those occupants

major failure conditions which reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a significant reduction in safety margins or functional capabilities, a significant increase in crew workload or in conditions impairing crew efficiency, or discomfort to occupants, possibly including injuries.

minor failure conditions which would not significantly reduce aircraft safety, and which would involve crew actions which are well within their capabilities. Minor failure conditions may include, for example, a slight reduction in safety margins or functional capabilities, a slight increase in crew workload, such as, routine flight plan changes, or some inconvenience to occupants.

no effect failure conditions which do not effect the operational capability of the aircraft or increase crew workload.

An error in software may cause a fault that contributes to a failure condition. Thus, the level of software integrity necessary for safe operation is related to the system failure conditions. For each software component or function in the system, a software level is defined. The level is defined based upon the highest categorisation of the failures of a system function which that software component or function may cause or contribute to as a result of anomalous behaviour. The software levels correspond to the failure condition categories as follows:

Level A catastrophic

Level B hazardous / severe-major

Level C major

Level D minor

Level E no effect

The standard states that the system safety assessment process determines the software level. This is the safety assessment performed on the system of which the software is a part, which identifies the failure conditions of the system components which may contribute to system hazards. A crucial point to note here is that the software level is qualitative, that is that the level relates to failure conditions, not to failure rates. Indeed the standard cautions that, "development of software to a software level does not imply the assignment of a failure rate for that software. Thus, software levels or software reliability rates based on software levels cannot be used by the system safety assessment process as can hardware failure rates." The reason for this is the systematic nature of software failures, which are not random and predictable like hardware failures. Software levels are therefore used in DO-178B as a way of ensuring the level of rigour used in developing the software is commensurate with the failure classification.

It is possible to use architectural means to reduce the software level of a particular component. For example by having multiple versions of dissimilar software providing the same function it

is possible for each of the components to be individually developed to a lower level. There are many issues to consider in doing this, such as ensuring independence between versions. This is not considered in detail here, as architectural issues do not impact on the analysis process developed in this thesis.

The standard provides guidelines on the objectives and outputs required for each software level at each stage of the software life cycle process. The software lifecycle is split into the software planning process, the software development process (which includes requirements, design, coding and integration), and the integrating processes (including verification, software configuration management, quality assurance, and certification liaison). At each stage the objectives are usually more onerous for the software of higher software levels. Some of the objectives may not need to be satisfied for lower level software, and some objectives may need to be satisfied with independence for higher levels. As an example, an extract from the table of the objectives for verification of outputs of software coding and integration processes is reproduced from the standard in figure 2.1.

	Objective		Applicability by SW Level				Output	
	Description	Ref.	A	B	C	D	Description	Ref.
1	Source code complies with low-level requirements	6.3.4a	●	●	O		Software Verification Results	11.14
2	Source code complies with software architecture	6.3.4b	●	O	O		Software Verification Results	11.14
3	Source code is verifiable	6.3.4c	O	O			Software Verification Results	11.14
4	Source code conforms to standards	6.3.4d	O	O	O		Software Verification Results	11.14
5	Source code is traceable to low-level requirements	6.3.4e	O	O	O		Software Verification Results	11.14
6	Source code is accurate and consistent	6.3.4f	●	O	O		Software Verification Results	11.14
7	Output of software integration process is complete and correct	6.3.5	O	O	O		Software Verification Results	11.14

LEGEND: • The objective should be satisfied with independence
o The objective should be satisfied
Blank Satisfaction of objective is at applicant's discretion

Figure 2.1: Verification of outputs of software coding and integration processes from standard DO178-B

The basic philosophy of this standard, as described here, is that the more severe the failure categorisation relating to the software, the higher the software level will be, and thus the more onerous will be the objectives for the process, and the better (i.e. safer) the resulting software will be. This type of approach, whereby low-level prescriptive objectives are placed on the developers is common to many software safety standards, and is often referred to as a *process*

driven approach. This is because the safety of the software is assured predominantly through appealing to the quality of the process used in developing the software.

The Federal Aviation Authority (FAA) have produced a guidance document [2] on how to comply with objectives of DO-178B when using OO technologies. This is achieved by looking at a number of issues associated with OO which could cause problems when trying to show compliance with DO-178B requirements. The features of OO which are addressed are such things as single and multiple inheritance, dynamic dispatch and reuse. These features are discussed in more detail later. The guidance given in [2] takes the form of a set of rules which should be adopted in the design and implementation in order to avoid common errors and pitfalls. Two examples of the rules provided under inheritance are: “To ensure that overriding is always intentional rather than accidental, design and code inspections should consider whether locally defined features are intended to override inherited features with a matching signature”, and “No overridden method should be called during the initialisation (construction) of an object”. This document provides a thorough and easy to use set of guidelines on good practice in OO development. The document is, however, very much focussed on the accuracy and reliability of the software. There is little guidance provided on a coherent safety process.

2.2.1.2 IEC 61508

IEC 61508 [23] is not specific to any particular industry and is intended to provide a unified and consistent approach to all the phases of the safety lifecycle for all types of component. Part 3 of the standard deals specifically with safety-related software. The standard uses the concept of safety integrity levels (SILs) to signify the safety integrity requirements for the functions performed by the system. SILs are similar to the software levels used in DO 178-B. There are four SILs, from 1 which is low, to 4 which is high. For software, the safety integrity is used as a way of avoiding “systematic failures in a dangerous mode of failure”. SILs are assigned using the severity of failure of the software functions. The standard requires that software functional safety requirements are considered, along with safety integrity requirements. The functional requirements define the required behaviour of the software for the safe operation of the system. The safety integrity requirements capture the degree of confidence required in these functions performing correctly. As such, it could be said that IEC 61508 is primarily concerned with ensuring the software is developed correctly in the first place, rather than finding errors later through verification. The SIL again determines the techniques that are required to be undertaken during development, with techniques (such as partitioning) being

either recommended, or highly recommended.

2.2.1.3 Def. Stan. 00-55 and 00-56

There has recently been a major shift in the standards used for safety related software in the U.K. defence industry. Previously, the standard used for the development of safety-related software was Def. Stan. 00-55 issue 2 [51]. This standard fell within the framework defined by Def. Stan. 00-56 issue 2 [49], which covered safety management of the entire system. Def. Stan. 00-55 was very similar to DO 178-B and IEC 61508 in its approach, in that a SIL is assigned to software components or functions dependent upon the criticality of a failure in the software. The safety integrity level then drives the design, development and assessment activities.

Def. Stan. 00-55 did have a number of strengths, for example the overarching system safety framework from 00-56 issue 2 helped to ensure that the software safety process was linked to the system safety process. One way in which Def. Stan. 00-55 differed from DO 178-B and IEC 61508 was in the requirement to produce a software safety case, which the standard defines as "a well-organised and reasoned justification, based on objective evidence, that the software does or will satisfy the safety aspects of the Software Requirement."[51] Software safety cases shall be discussed in more detail in section 2.2.3.

Def. Stan. 00-55 and other defence safety standards have now been superseded by issue 3 of Def. Stan. 00-56 [52], which takes a fundamentally different approach to the other standards discussed here. There has been much debate over many years within the software safety community as to the effectiveness of the so called process-based approaches taken by the standards. In [45], McDermid examines the evidence to support the assumption that the processes prescribed for software at higher integrity levels necessarily produces "better" software. Although acknowledging that the assessment cannot be taken as conclusive, he determines that at a minimum the assumption seems questionable. He suggests that what evidence there is points to the most significant factor in achieving low hazardous failure rates is the level of domain knowledge of the developer, rather than the development process or language. A suggested reason for this is that while the standards contain much useful advice, their focus is actually on the reliability of the software, rather than safety. McDermid therefore suggests that an alternative approach be taken, which is "to seek explicit evidence of safety, with respect to potentially hazardous failure modes of the software, rather than make a "general appeal" to the quality of the process" [45].

Def. Stan. 00-56 issue 3[52] has adopted this philosophy, and thus the production of a safety case is key to compliance with the standard. The safety case must be produced for all systems

including software systems developed under this standard. The safety case is a structured argument, supported by a body of evidence, that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given environment. The concept of a safety case is explored in more detail in section 2.2.3 of this literature survey. The standard identifies that a major part of the safety case will be the output from a risk management process. This process controls the risk associated with the hazards identified for the system under consideration. Risk is defined as a "combination of the likelihood of harm and the severity of that harm"[52] and for each system hazard should be reduced to a broadly acceptable level where possible, and where this is not possible, risks should be reduced to levels that are tolerable and as low as reasonably practicable. The techniques used in order to manage the risks, and develop a safe system, are not mandated by the standard (although guidance is provided for developers of safety critical systems). The onus is therefore on the software system developer to identify and use the appropriate techniques as part of a software safety development process, and to demonstrate how they contribute to the safety of the resulting system. The software safety lifecycle is discussed in detail in section 2.2.2.1. The focus of the approach taken in Def. Stan. 00-56 issue 3 is not to look at the process used to develop the software, but to focus on generating evidence that directly supports the safety of the software product being developed. Thus the approach is often referred to as a *product-driven approach* (or *evidence-based approach*) to software safety.

2.2.1.4 Advantages of a product-based approach

With the more prescriptive standards (such as Def. Stan. 00-55 discussed above), the developer has less freedom to select the evidence necessary to support the claims made about the safety of the system than with a product-based approach. The product-based approach potentially requires additional work on the part of the developer to identify suitable evidence to support the safety case (as this is no longer explicitly defined in the standard). However this also means that techniques inappropriate to the system being developed do not need to be undertaken, providing that the evidence can be gained from a different source and that this can be justified in the safety case. This can be particularly useful for the developer of a system that uses new technologies, where existing approaches to assuring the safety of the system may not be appropriate. The freedom offered by the product-based approach allows the developer to exploit different analysis or verification techniques in order to demonstrate the new technology may be exploited safely. This is thus a useful context in which to consider the development of safety

critical OO software.

It is also worth noting here that a product-based approach does not exclude appeals to process quality as part of the safety argument, nor are the approaches prescribed in other standards discouraged. Indeed if a process based standard can be shown to provide the direct evidence of the safety of the system required, then this could be used in developing and justifying the safety of the system. Crucially, it is not enough simply to show that that process has been followed. The way in which that supports the safety claims made about the system must also be made clear.

2.2.2 Safety critical system development

"Safety is a property of a system."[74] A system can be considered safe if all the hazards have been eliminated, or the risks associated with those hazards have been reduced to an acceptable level. Software forms part of a system, and can only be considered safe (or indeed unsafe) within the context of that system. Software may operate safely within one particular system, that is to say that the operation of that software does not contribute to any hazardous behaviour. However, if that software were to be used in a different system, it may function in exactly the same manner as it had done in the original system (where its behaviour had been safe) and yet contribute to hazardous behaviour. This was made clear by the Ariane 5 accident, where software which had been used safely in a previous system caused a hazardous failure when used within Ariane 5. Since each system is different, with different requirements, hazards and characteristics, it is impossible to know if software is safe without considering the behaviour of the software as part of the system which it is controlling. Therefore when considering the process for developing safe software, it is crucial that the whole of the system of which the software is a part is considered, as well as the software itself. This survey shall review the techniques available for developing safety critical systems.

2.2.2.1 The software safety lifecycle

There are various activities which need to be performed in order to ensure the safety of a system. These activities are carried out at different stages throughout the development lifecycle of the system from initial requirements analysis, through to delivery of a completed system (and indeed beyond into maintenance activities). Figure 2.2, adapted from [64] and the safety assessment process of ARP 4754 [55] shows a traditional simplified V lifecycle model for the development of a system. Outside of this V lifecycle, different stages of the safety process have been identified

which are associated with the various development steps of the lifecycle. This review identifies the purpose of each of the stages, as well as the safety analysis techniques that are available at each analysis stage. Although focussing on software safety techniques, the review will also consider analysis techniques which are not specific to software but are crucial to the development of any safe system.

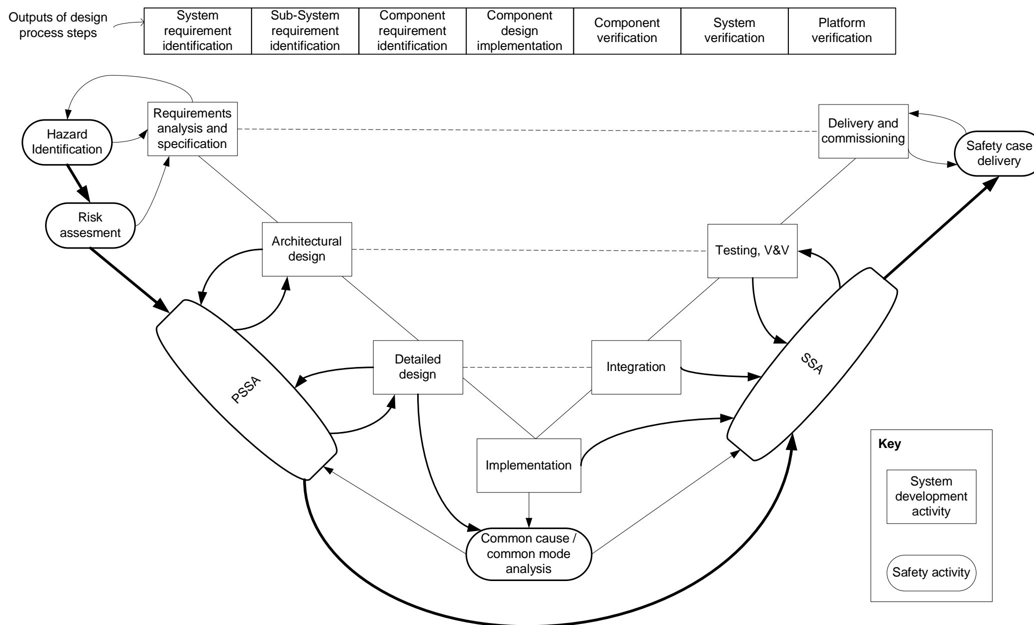


Figure 2.2: V lifecycle model showing safety activities

Figure 2.2 also identifies the output from each step of the design process. It can be noticed from this that the process is decompositional in nature. Figure 2.3, taken from [80] illustrates this decomposition. Initially the highest system level is considered, which could for example be an aircraft, a train, or a nuclear reactor. As a more detailed architecture is produced, the sub-systems which make up that platform are considered. As the design becomes more detailed the components which make up those sub-systems can also be considered. It is most likely to be at the component level that the software aspects of the design will emerge. The decomposition may continue, as relevant to the system under development, as more detail becomes available. The safety of the design must be considered at each stage in the decomposition, and the results of the analysis must also be decomposed as the design becomes more detailed.

2.2.2.2 Hazard identification

The objective in developing a safe system is to ensure that all the hazards are managed and controlled. Therefore the first step in the safety process is to identify what those hazards are.

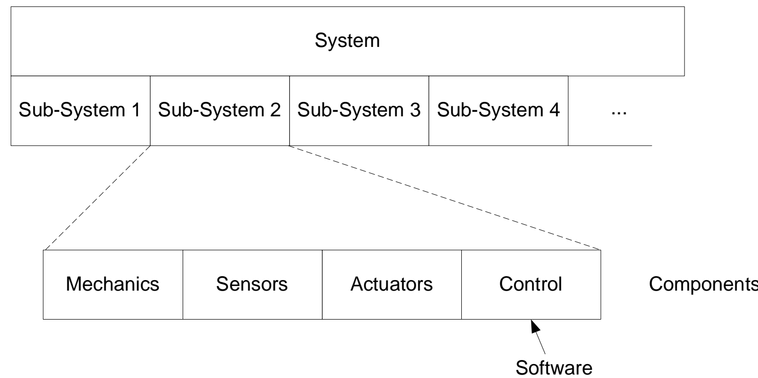


Figure 2.3: The decomposition of a platform design

Hazard identification, often referred to as Preliminary Hazard Identification or PHI, occurs at the very start of the development process when the requirements for the system are being developed. It is the hazards identified at this stage which ‘drive’ the rest of the safety process. In practice the hazards associated with most types of system are fairly well known as most safety-critical systems are developed in domains where the developers have a lot of previous experience of developing similar systems. As a result there are few techniques other than brainstorming, or the use of checklists available for conducting PHI. Identifying novel hazards for a system will normally involve people with domain expertise considering “what if” scenarios.

Despite the lack of techniques available for this stage of the safety process, its importance should not be undervalued. If some system hazards are missed, or incorrectly identified this will have a huge effect on the effectiveness of the rest of the process.

2.2.2.3 Risk assessment

Risk assessment looks at each of the hazards identified for the system and determines the risk associated with that hazard. Risk is a measure of the threat posed by a hazard, and is defined in Def. Stan. 00-56 issue 3 [52] as the “combination of the likelihood of harm and the severity of that harm”. By assessing the risks it is possible to make decisions on the steps that will be necessary in order to manage the hazard by reducing the risk to an acceptable level.

Risk assessment is normally carried out in a qualitative, rather than a quantitative manner, as data for a qualitative assessment is often unavailable. As risk is a product of two factors a matrix is often used to identify the risk. Figure 2.4 shows a risk assessment matrix taken from [57] which categorises the likelihood of occurrence from frequent to improbable, and the severity from catastrophic to negligible. The risk for each combination of likelihood and severity

is denoted as either high, medium or low.

		Hazard Severity Categories			
Frequency of Occurrence		I CATASTROPHIC	II CRITICAL	III MARGINAL	IV NEGLIGIBLE
A	**FREQUENT**	HIGH	HIGH	HIGH	MEDIUM
B	**PROBABLE**	HIGH	HIGH	MEDIUM	LOW
C	**OCCASIONAL**	HIGH	HIGH	MEDIUM	LOW
D	**REMOTE**	HIGH	MEDIUM	LOW	LOW
E	**IMPROBABLE**	MEDIUM	LOW	LOW	LOW

Figure 2.4: Risk assessment matrix

The risk associated with each of the hazards should be reduced to an acceptable level. This can be done by:

1. Eliminating the hazard
2. Reducing the probability of, or mitigating the effects of the hazard
3. Providing alerts and warnings
4. Establishing procedures

This list of measures is given in order of priority, with eliminating the hazard being the most preferable, and establishing procedures being the least. The level of risk which is deemed to be acceptable for a hazard must be determined, taking into account that it is not always feasible to reduce the risk of all hazards to a low level. In such cases the acceptance of higher risk must be justified.

When reducing the risk of a hazard, an approach which is often used in the UK is to show that the hazard has been reduced As Low As Reasonably Practicable (ALARP). The ALARP principle, which has emerged from principles enshrined in the Health and Safety at Work Act [22], considers there to be three risk regions. The first region is described as *broadly acceptable*, and contains hazards of low-level risk for which no additional reduction is required. Another region is described as *intolerable*, and any hazards in this region must be eliminated. Between these two regions is the *ALARP region*. Hazards whose risk falls in this region can only be considered acceptable if it can be shown that the cost of reducing the risk further is grossly disproportionate to the improvement that is gained. Defining the limits between the regions is a key aspect of ALARP.

The risk assessment is used in defining the system level safety requirements. These requirements form a key part of the risk reduction strategy for those hazards whose risks were assessed as unacceptable. The risk assessment process can also be used to identify the hazards with the higher risk which may require more attention or resource assigned to them.

2.2.2.4 Preliminary System Safety Assessment

Once the design activities commence, the aim of the safety analysis becomes ensuring that a proposed design can meet its safety requirements, and also to refine the safety requirements where necessary. This part of the analysis process is referred to as Preliminary System Safety Assessment (PSSA). The techniques used for PSSA aim to start from the system level requirements identified in the hazard analysis and risk assessment, and investigate the possible causes of failure modes in the proposed design. As the design becomes more detailed at the component level, the derivation of more detailed safety requirements to prevent or control the failure modes are derived. It is at this stage in the process that the software components within the system start to be considered. Although the aim of the PSSA activities is the same for software components as for other more traditional components, many of the analysis techniques cannot be directly applied to software, so need to be adapted, or alternative techniques used. This review will particularly focus on the analysis techniques for software.

Fault tree analysis

Fault tree analysis is by far the most common PSSA analysis technique. A fault tree is used to represent in a graphical form how combinations of events can contribute to some top event, which would be the undesirable failure event of interest. The main elements of the notation, taken from [58], are shown in figure 2.5.

In addition to those shown, other symbols are available such as the exclusive OR gate, and rather more exotic variants such as the summation and delay gates suggested by Villemeur [81] however in many cases, the basic symbol set is sufficient. A very simple example taken from [74] is shown in figure 2.6.

It will be noted that in the example a clear distinction is made between primary and secondary failures of components. Failures can also be classified as command failures. Identifying the type of failure can be very useful in fault tree analysis. Villemeur [81] classifies the failures as:

Primary failure Failure of an entity not caused by the failure of another entity. For a compo-

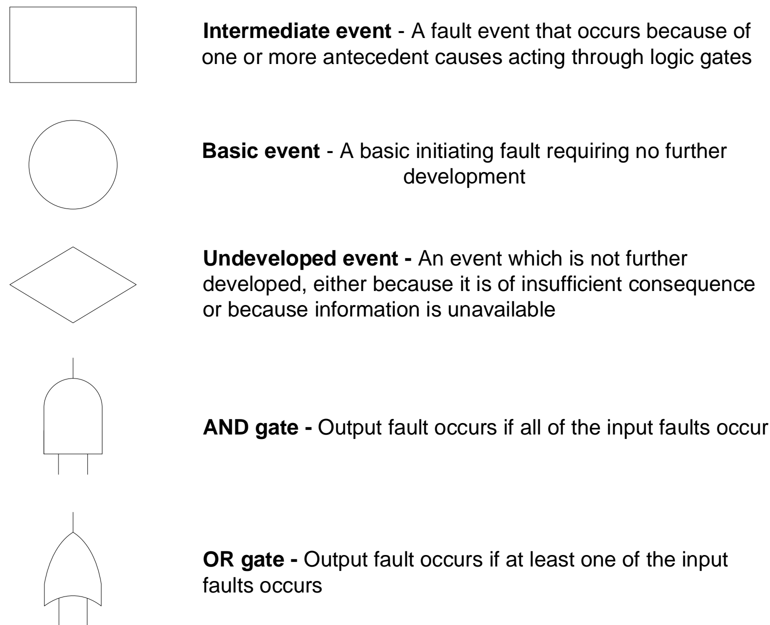


Figure 2.5: Main elements of the fault tree notation

nent under operation, the failure may, for instance, originate in wear problems or defects in its design.

Secondary failure Failure of an entity caused by the failure of another entity this entity was neither qualified for nor designed against. Particular conditions in the environment or human errors may result in the secondary failure of a component.

Command failure Failure of an entity caused by the failure of another entity this entity was qualified for or designed against. Such a failure occurs when the entity changes state following inadvertent control signals.

The technique for constructing fault trees generally takes the form of sets of guidelines rather than any formal rules. Both [58] and [81] contain rules for fault tree construction.

Fault trees can be used to derive safety requirements for the design, by identifying potential failures that can lead to the hazardous failure mode at the top event. Fault trees can also be used to calculate probability requirements for particular failure events. If the required probability for the top event is known, then this probability can be propagated down the tree. At each level a decision is made on how to allocate the probability between the events at that level. This continues down the tree until probabilities can be allocated to the basic events. These probabilities then become targets as part of the requirements relating to that failure i.e. these probabilities are derived safety requirements upon the system.

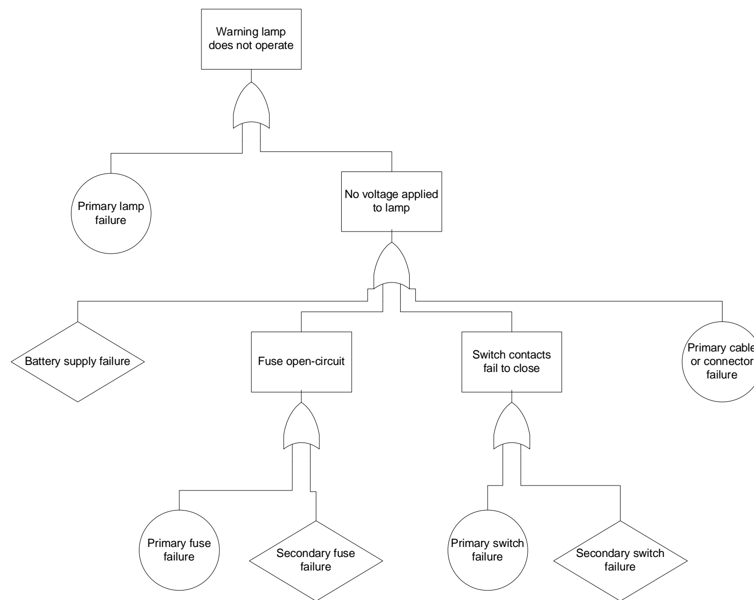


Figure 2.6: A simple fault tree

These standard fault trees could be used to identify failures of software components in the system design. A command failure in a fault tree for example could be that a software component generates an undesirable output condition. In this way requirements may be defined upon the software component. As will be seen in the discussion of system safety analysis, fault trees can also be used as a confirmatory technique once a design has been implemented. To use fault trees for confirmatory analysis of software it is necessary to consider the logic contained within the program. It shall be seen later how fault trees can be extended such that the design of the software itself may be considered.

HAZOP

Hazard and operability studies, or HAZOP as they are more commonly known, were originally developed by ICI in the 1960's as a way of analysing process plants [9], and is still very widely used today, especially in the chemical and nuclear industries. The reason for its popularity in such industries is that HAZOP, rather than concentrating on the failure of components, instead focusses on the behaviour of flows between components. HAZOP is a technique that aims to predict possible failures, and identify their impact. HAZOP is referred to as an imaginative anticipation of failures, and therefore takes the form of a group brain-storm style activity. A set of guidewords are used to prompt the identification of deviations from normal behaviour for a particular flow in the design. The possible causes and consequences of each deviation are

considered, and if the deviation has a plausible cause and could lead to a hazardous failure mode then actions are suggested and derived safety requirements generated. Figure 2.7 taken from [64] shows the standard HAZOP guidewords with some example interpretations.

Guide Word	Deviation	Example Interpretation
NO or NONE	No part of the intention is achieved	No forward flow when there should be.
MORE	Quantitative increase in a physical property (rate or total quantity)	Higher pressure, flow rate, temperature… Quantity of material is too large.
LESS	Quantitative decrease in a physical property (rate or total quantity)	Low pressure, flow rate, temperature… Quantity of material is too small.
MORE THAN or AS WELL AS	All intentions achieved, but with additional effects (qualitative increase)	Impurities in flow (air, water, oil…) Chemicals present in more than one phase (vapour, solid)
PART OF	Only some of the intention is achieved (qualitative decrease)	One or more components of mixture missing, or ratio of components is incorrect.
OTHER THAN	A result other than the intention is achieved	Any state other than normal operation, e.g. startup, shutdown, maintenance…
REVERSE	The exact opposite of the intention is achieved	Reverse flow.

Figure 2.7: HAZOP guidewords

Although still predominantly used in process industries, it is possible for HAZOP to be adapted for other applications. A number of attempts have been made to adapt HAZOP for use in analysing software. This is probably best summed up in the now obsolete MOD Defence Standard 00-58 [50]. The main challenge in applying HAZOP to software is in interpreting the standard HAZOP guidewords in the context of the software design being analysed. Def. Stan. 00-58 includes guidance on guide word interpretations for different attributes of various software design representations including data flow diagrams, state transition diagrams and object oriented designs. The application of HAZOP to OO designs shall be looked at in detail in section 2.3. As an example of the way Def Stan 00-58 suggests interpreting HAZOP guidewords, figures 2.8 and 2.9 show the interpretations for data flow diagrams and state transition diagrams.

The SHARD (Software Hazard Analysis and Resolution in Design) analysis technique developed by Pumfrey [64] both influenced, and developed upon the approach described in Def Stan 00-58. In structuring the guidewords used in SHARD, the failure classes proposed by Bondavalli and Simoncini in [6] were used. These failure classes are applied to each information flow in a software design. Each information flow is considered as a separate *service*. Figure 2.10 taken from [64] shows failures defined for each service group.

Classifying failures in this manner helps to ensure that the possible failures for each flow are considered. The failure classes are each interpreted for the information flow being considered. Figure 2.11 shows how the guidewords could be interpreted for various flows in the MASCOT 3 design notation [26].

Attribute	Guideword	Interpretation
Flow (of data or control)	No	No information flow
	More	More data is passed than expected
	Part of	The information passed is incomplete (for group flows)
	Reverse	Flow of information in wrong direction (normally not credible)
	Other than	Information complete, but incorrect
	Early	Flow of information occurs before it was intended
	Late	Flow of information occurs after it was required
Data rate	More	The data rate is too high
	Less	The data rate is too low
Data value	More	The data value is too high (within or out of bounds)
	Less	The data value is too low (within or out of bounds)

Figure 2.8: Example attribute guide word interpretations for data flow diagrams

Similar interpretations of the guidewords for other software design notations can be produced.

Functional Failure Analysis

Functional failure analysis (FFA) is a predictive analysis technique which focusses on the *functions* of a system. In a similar manner to HAZOP, FFA uses hypothetical failure types as prompts for identifying failure modes. FFA is very popular in the aerospace industry. The best description of the technique is found in ARP 4761 [56] where it is referred to as Functional Hazard Assessment. For FFA the failure types are:

- Function not provided
- Function provided when not required
- Function provided incorrectly

When performing FFA, Pumfrey [64] identifies the following steps:

- Identify functions
- For each function identified, suggest failure modes, using the three failure types as prompts
- For each failure mode consider the effects
- Identify and record any actions necessary to improve the design

Attribute	Guideword	Interpretation
Event	No	Event does not happen
	As well as	Another event takes place as well
	Other than	An unexpected event occurs instead of the anticipated event
Action	No	No action takes place
	As well as	Additional (unwanted) actions take place
	Part of	An incomplete action is performed
	Other than	An incorrect action takes place
Timing of Event or Action	No	Event/action never takes place
	Early	Event/action takes place before it is expected
	Late	Event/action takes place after it is expected
	Before	Happens before another event or action that is expected to precede it
	After	Happens after another event or action that is expected to come after it

Figure 2.9: Example attribute guide word interpretations for state transition diagrams

Group	Failure classes
Service provision	Omission, Commission
Service timing	Early, Late
Service value	Coarse (detectably) incorrect Subtle (undetectably) incorrect

Figure 2.10: Failure classifications used to structure SHARD guidewords

One of the key features of FFA is that it can be applied at any level of design detail. This makes it ideal for establishing safety requirements and propagating those requirements down from high level design to more detailed component design.

FFA is widely applied to software designs. The most likely reason for its popularity is the ease with which the standard FFA technique can be applied directly to software functions. The only real problem that arises seems to be that interpreting 'function provided incorrectly' for software can be difficult. It is usually interpreted as meaning that a wrong value is provided by the function, however this potentially misses other important failure modes which may be covered by this prompt. Another weakness of this approach is that only the effects of a failure are considered, not the causes. The result of this is that every potential failure is considered to be a valid one, which is obviously overly pessimistic.

Flow		Failure Categorisation					
		Service Provision		Timing		Value	
Protocol	**Data Type**	**Omission**	**Commission**	**Early**	**Late**	**Subtle**	**Coarse**
Device input	**Boolean**	No read	Unwanted read	Early	Late	Stuck at 0 Stuck at 1	N/A
	Value	No read	Unwanted read	Early	Late	Incorrect in range	Out of range
Device output	**Value**	No write	Unwanted write	Early	Late	Incorrect	N/A
Pool	**Enumerated**	No update	Unwanted update	N/A	Old data	Incorrect	N/A
	Value	No update	Unwanted update	N/A	Old data	Incorrect in range	Out of range
	Complex	No update	Unwanted update	N/A	Old data	Incorrect	Inconsistent
Channel	**Boolean**	No data	Extra data	Early	Late	Stuck at 0 Stuck at 1	N/A
	Complex	No data	Extra data	Early	Late	Incorrect	Inconsistent

Figure 2.11: Example SHARD guidewords for MASCOT 3

Summary of PSSA techniques

The review has looked at some of the most commonly used techniques for PSSA: Fault trees, HAZOP and FFA. The aim of all the techniques is essentially the same, that is to identify failures which could lead to a system hazard so that safety requirements may be derived which, when implemented, will prevent the occurrence of the hazard. Fault trees focus on component failures and is a deductive technique, that is to say that it works back from a hazardous failure event to identify possible causes. HAZOP considers failures relating to flows between components, and is an inductive technique. That is to say that failures in the behaviour of the flows are postulated, and the consequences of the failure are considered. FFA is also an inductive technique, however it considers failures of functions. There are also a number of other techniques which are used in PSSA for analysing failures, an overview can be found in [40]. It is impossible to say that any particular technique is ‘better’ than any other, as different techniques work best with different types of system or component. It is important the the most appropriate technique, or combination of techniques is selected.

All of the techniques reviewed here have the ability to be applied to software, with varying levels of success and ease. Which of the techniques it is preferable to use will depend upon the software system itself, and its design. For some software, a functional view may be the most appropriate, in which case FFA may prove the most effective. For other software designs, it may be the information flows which are most important in defining the software, and therefore HAZOP or SHARD would be more appropriate.

There have been a number of attempts to develop techniques specifically for the analysis of OO software. These shall be reviewed later when discussing the object oriented paradigm in more detail.

2.2.2.5 System Safety Analysis

Once the design is implemented, it is necessary to confirm that the safety requirements derived during the PSSA stage of the analysis process have been met. The aim is to generate evidence that can be used in the safety case for the system. There are two main techniques which are used for system safety analysis (SSA) of conventional systems: Fault trees, and failure modes and effects analysis (FMEA). Here these techniques are briefly discussed, before looking in more detail at how SSA is carried out for software.

Confirmatory fault tree analysis

When the use of fault trees was discussed previously, their purpose was as an aid in identifying potential causes of failures as part of PSSA activities. In this role fault trees are a predictive analysis technique. Fault trees can also be used in a confirmatory role where they are particularly useful in showing that a probability requirement for a hazardous failure mode has been met by the system. The *known* (or estimated) failure probabilities for the basic failure events can be included in the fault tree, and then the probabilities of the intermediate events can be calculated. This will eventually lead to an 'achieved' probability for the top event. It can then be seen whether the required probability has been met. To assist in this analysis it is possible to reduce a fault tree to its minimal cut set form. The fault tree handbook [58] defines a minimal cut set as "A smallest combination of component failures which, if they all occur, will cause the top event to occur."

It is also possible to use fault trees for system safety analysis of software. Leveson [38], [39] has developed an approach know as Software Fault Tree Analysis (SFTA) which is used as a confirmatory analysis to show that the software program produced does not lead to hazardous failure modes. Essentially SFTA is a form of static code analysis (see section 2.3.6), albeit a very hazard focussed form. SFTA is carried out by inspecting the logic of the program code relating to the hazardous failure mode of interest. A fault tree is constructed to show what steps in the program code would need to fail to lead to the top event. Failure events that could not be produced by the code are eliminated until either the entire tree is eliminated (indicating the

code could not lead to the top event) or the faults in the code are identified. The main problem with the SFTA approach is that it can only really be used effectively on quite small amounts of code. For larger programs it is probably more cost effective to use static code analysis.

Failure Modes and Effects Analysis

It was seen above how fault trees can be used in both a predictive, and a confirmatory role. Failure Modes and Effects Analysis (FMEA) can similarly be applied as a technique for both PSSA and SSA. FFA, discussed as part of the PSSA section, is essentially a predictive FMEA. FMEA itself is a confirmatory analysis, which uses the known failure modes of system components for a specific system implementation to determine which failures may occur at the system level. The best descriptions of the FMEA technique can be found in [81] and [56]. Villemeur identifies four main steps in performing an FMEA:

1. Definition of the system, its functions and components;
2. Identification of the component failure modes and their causes;
3. Study of the failure mode effects;
4. Conclusions and recommendations

In identifying the failure modes for each of the components, it is possible to make use of guide lists of generic failure modes, however specific failures for each component should be identified through, for example, operating experience, or the use of component testing. Figure 2.12 taken from [81] shows the typical output for a simple example FMEA represented in tabular form.

FMEA is often a technique which is concerned with system reliability, rather than system safety. FMEA can be used quite effectively for safety however, in which case it is important to identify which of the effects are hazardous, and what mitigations are in place for each hazardous effect. In this way the FMEA is used to provide evidence that the safety requirements identified at PSSA stage are met by the implemented system. An adaptation of FMEA which takes more account of these safety issues is failure modes, effects and criticality analysis (FMECA). This technique incorporates the risk associated with each failure mode into the FMEA output. In this way it is possible to check that the risks associated with the failure modes are at an acceptable level. If there are failure modes whose risks are unacceptable, then further measures are required to mitigate those risks.

Components	Failure modes	Possible causes	Effects on the system
Push –button (P.B.)	• The P.B. is stuck • The P.B. contact remains stuck	• Primary (mechanical) failure • Primary (mechanical) failure • The operator fails to release the P.B. (human error)	• Loss of the system function: the motor does not operate •The motor operates too long: hence a motor short circuit, which leads to a high electric current and the melting of the fuse
Relay	• The relay contact remains opened • The relay contact remains stuck	• Primary (mechanical) failure • Primary (mechanical) failure • A high current passes through the contact	• Loss of the system function: the motor does not operate • The motor operates too long: hence a motor short circuit, which leads to a high electrical current and the melting of the fuse
Fuse	• The fuse does not melt	• Primary failure • The operator overrated the fuse (human error)	• In case of a short circuit, the fuse does not open the circuit
Motor	• The motor does not operate • Short circuit	•Primary failure •The P.B. is stuck • The relay contact remains opened • Primary failure •The motor operates too long	• Loss of the system function: the motor does not operate •The motor short circuit causes a high electric current to pass and then the melting of the fuse; the relay contact remains stuck

Figure 2.12: An FMEA table

FMEA is not really a technique that can be used for SSA of software. Although software components could be considered in the FMEA, meaningful failure modes would be difficult to determine. The FMEA technique is simply not fine-grained enough for confirmatory analysis of software code. Software FMEA (SMEA) [65], [36] has however been used with some success during requirements and design analysis [43]. There are a number of limitations to the technique however. Lutz [43] describes the technique as time-consuming and tedious, and also identifies that its success depends upon both the domain knowledge of the analyst and the accuracy of the documentation. Lutz also confirms that a complete list of failure modes for software cannot be compiled. She identifies however that by integrating SFMEA with a backward search for contributory causes, such as FTA, the effectiveness of SFMEA can be enhanced.

Software verification

There are many software verification techniques which can be used to generate evidence that the software meets its requirements. The generation of evidence is an important aspect of the safety process, however one of the objectives of the thesis is that, as far as possible, the approach proposed will fit in with existing development processes. It is therefore not the intention of this thesis to attempt to propose any new verification techniques, nor indeed does it particularly aim to comment on the merits of any specific existing techniques. However, if the existing verification techniques are to be used effectively as part of the approach developed in this thesis, then it is important to provide an overview of those verification techniques, and in

particular, techniques for verifying OO software. Verification techniques for OO are discussed in more detail in section 2.3.6. Here a few general observations about software verification are made.

In verifying the *safety* of software, it is not good enough just to show that the implementation satisfies the functional requirements, nor is it even in some senses *necessary* to show this. As Leveson points out, "Not all software errors cause hazards, and not all software that functions according to specification is safe". The crucial thing for safety is that the software is verified specifically against the safety requirements. Verification is generally split into two types of analysis: dynamic analysis, and static analysis. Dynamic analysis involves executing the code and evaluating its behaviour. Static analysis involves evaluating the code without executing it. There are limitations to both types of analysis which will not be discussed here, however when verifying the safety of software a combination of both static and dynamic techniques is usually appropriate (this is mandated by most standards).

2.2.2.6 Common Cause Analysis

Common cause analysis, or common mode analysis as it is also referred to, is used throughout the whole development lifecycle in support of both PSSA and SSA. Many of the analysis techniques which have been discussed here, such as fault trees, assume that failures are independent of each other. If this assumption is not true, and there in fact exists a common cause, then the fault tree is invalid. If two basic failures below an AND gate are considered, the probability of the intermediate failure should be the product of the two basic failure events. If the two failures are not independent, then the probability of the intermediate failure will be much higher than had previously been predicted. Identifying common causes can be done using a number of techniques.

ARP4761 [56] identifies three common cause analysis techniques: common mode analysis (CMA), particular risk analysis (PRA), and zonal safety analysis (ZSA). CMA uses checklists to identify potential sources of common mode failures in a design. These common failures may be design flaws common to a number of components, or external threats such as fires. The output of the PSSA analyses is used to determine where independence is required in the design and based on these two steps CMA requirements are derived. The design is then analysed to ensure the derived CMA requirements are met. If the CMA requirements cannot be met with the current design then modifications are required to resolve this. Software is identified in ARP4761 as an example of a potential cause of common mode failures. Systematic errors may well be common

to many items of software, particularly if the software items were developed by the same development team. This is particularly important where redundancy is claimed in software, as programming errors may be common to both items. It must be remembered that software can also be susceptible to external threats, as if the hardware on which the software runs fails, then so does the software.

PRA is used to analyse in detail the potential effects of events which are outside the system, but which could impact independence claims made about the system. ARP4761 suggests particular risks relevant to aircraft systems such as fire, tyre burst and bird strike. The effect of each particular risk on all parts of the system is evaluated in detail. ZSA is a technique which considers the effect of the physical proximity of components in causing common failures. This is done by splitting the system into zones, where each zone is separated by some form of containment. The sources of failure in each zone are identified along with the potential impact on the whole of the zone.

2.2.2.7 Safety case delivery

The final part of the safety lifecycle, as can be seen in figure 2.2 is to deliver a safety case for the system. Kelly defines a safety case in the following terms, “A safety case should communicate a clear, comprehensible and defensible argument that a system is acceptably safe to operate in a particular context” [29]. The safety case presents the evidence generated to support the safety of the system, and also an argument to show how that evidence provides support for the system’s safety requirements and objectives. As Kelly notes “evidence without argument is unexplained - it can be unclear that (or how) safety objectives have been satisfied”. Section 2.2.3 looks in detail at the production of safety arguments for software as part of a software safety case.

It should be noted that although the *delivery* of the safety case occurs at the end of the lifecycle, the actual process of generating the evidence to support the safety argument occurs from the very earliest stages of the development of the system. The output of the hazard identification and risk assessment activities for example would be expected to form part of the safety case evidence. It will often also be the case that interim safety case reports are produced at different stages in the development of the system.

2.2.2.8 Summary of safety critical systems development

This section has given an overview of the software safety lifecycle and some of the techniques which are used as part of the process. The way in which the different parts of the process fit together is neatly summed up in figure 2.13 taken from [64]. This shows how initially the causes and effects of failures in the system are unknown. Performing PHI identified hazards in the system. At this stage, the causes and effects are still unknown, however the effects have been projected (the outcome predicted) in the form of the hazards. Once PHI is completed the PSSA analysis begins. This aims to predict the causes of the hazards. PSSA leads to the central square of the matrix where projected causes and projected effects are both known. It is at this point that safety requirements will be put in place. These requirements are implemented within the design of the system. Once the detailed design is complete, it is possible to identify which of the projected causes are actually possible. Therefore the causes are now known. Similarly the effects are also known. Therefore it is possible at this stage to perform the confirmatory analysis of SSA, which checks the design's known behaviour meets its safety requirements. It is by linking the design back to a set of requirements generated from the system hazards that the safety of the system can be assured.

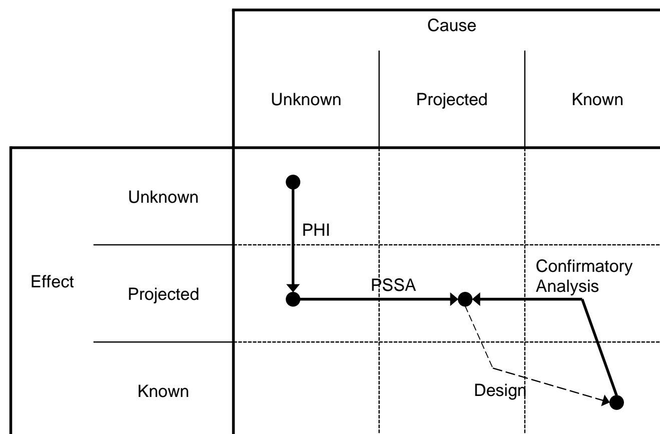


Figure 2.13: Representation of the safety critical systems development process

2.2.3 The software safety case

It was discussed in section 2.2.1 how there was a move towards a more product focussed approach to certifying safety critical systems, rather than appealing to process standards. This requires that a safety argument be produced which identifies the contribution of the software to the safety of the system. In [83] Weaver et. al. propose an approach for articulating software safety arguments which is largely independent of the development process. Their approach uses the

Goal Structuring Notation (GSN) to represent the arguments.

GSN is a graphical notation which can be used to record and present safety arguments [29]. GSN provides a notation for representing the elements of the argument and the relationship between those elements. The principal elements of the GSN notation are shown in figure 2.14. As described by Kelly in [31], "The principal purpose of a goal structure is to show how goals (claims about the system) are successively broken down into sub-goals until a point is reached where claims can be supported by direct reference to available evidence (solutions). As part of this decomposition, using the GSN it is also possible to make clear the argument strategies adopted (e.g. adopting a quantitative or qualitative approach), the rationale for the approach (assumptions, justifications) and the context in which goals are stated (e.g. the system scope or the assumed operational role)." Figure 2.15 shows an example goal structure in GSN taken from [83].

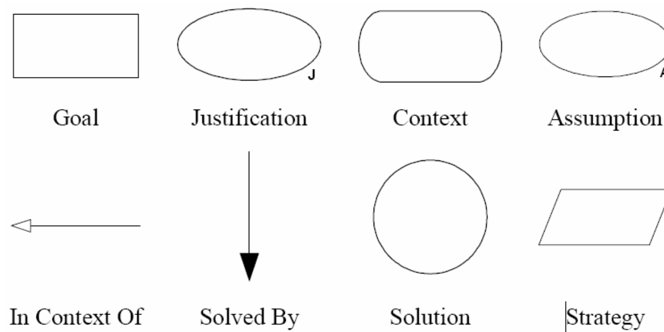


Figure 2.14: Main elements of the GSN notation

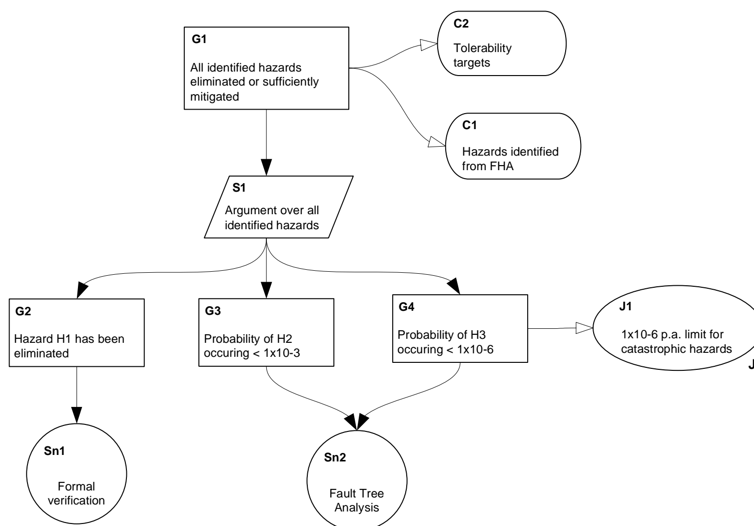


Figure 2.15: An example GSN goal structure

Kelly also introduces the concept of a safety case pattern as a way of explicitly capturing and

documenting reusable safety case elements. These reusable elements can then be instantiated for a particular implementation of a system to create a safety argument. In figures 2.16, 2.17 and 2.18 some of the safety argument patterns created by Weaver in [84] for arguing the safety of software are shown. These are generic patterns that can be used to create arguments about any safety critical software. The first pattern (figure 2.16) shows the top level decomposition for the safety argument of a system. This identifies the software as a contributor to system level hazards. Figure 2.17 then shows the pattern for decomposing the software contribution into a number of hazardous software failure modes (HSFM). The pattern in figure 2.18 then shows an approach to demonstrating that the causes of the HSFMs are acceptable by classifying the software failures into different types. These three patterns represent just one possible approach to a software safety argument, however they are useful in that they show that it is possible to construct a safety argument for software. It shall be seen later in the thesis that an alternative safety argument structure proves to be more effective for OO software. This is discussed in detail in chapter 5.

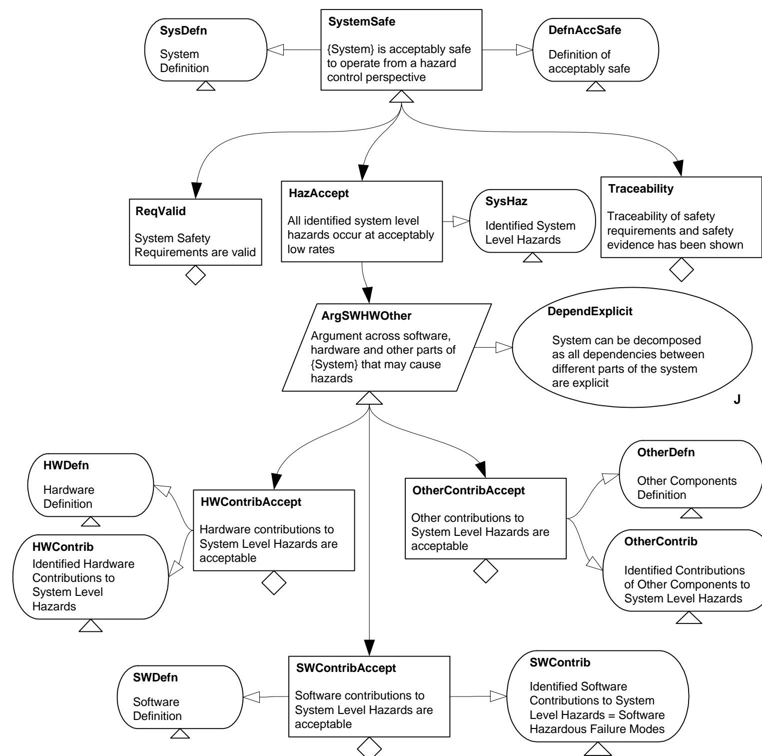


Figure 2.16: GSN Argument Pattern for Component Contributions to System Hazards

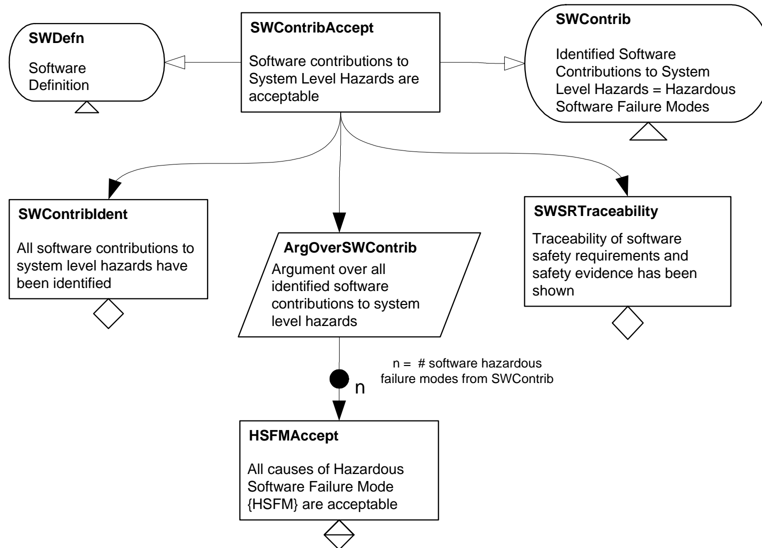


Figure 2.17: GSN Argument Pattern for Hazardous Software Failure Mode Decomposition

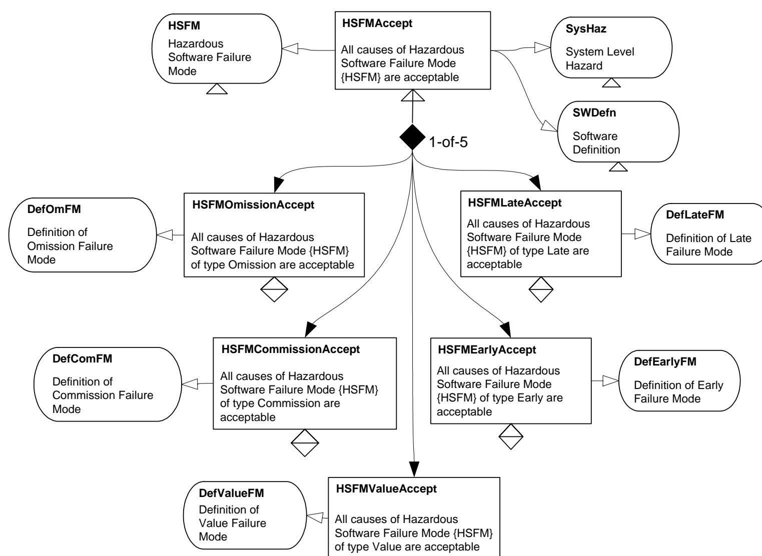


Figure 2.18: GSN Argument Pattern for Hazardous Software Failure Mode Classification

2.3 The object oriented paradigm

As with many of the developments within the computer science arena, when the object oriented paradigm first emerged it was purely in relation to programming languages. The work in this thesis is mainly focussed not upon OO programming (which is concerned with implementation), but with OO analysis and design. Section 2.3.2 will look at design and analysis methods for OO. Firstly section 2.3.1 introduces the basic concepts of OO.

2.3.1 Object oriented concepts

Simula 1 and Simula 67 were the first OO languages developed in the 1960s and contained most of the key OO concepts. These concepts are described below using definitions taken from [44] and [33].

The fundamental building block of all OO systems is the object itself. "An *object* is any thing, real or abstract, about which we store data and those methods that manipulate that data"[44]. Methods are also often referred to simply as operations and specify the way in which an object's data is manipulated. The methods therefore represent the behaviour of the object. A key aspect of methods is that an object's methods can only access the data of that object. Methods should not be able to directly access the data of another object. In order to access the data of another object, the object must send a message to that object. A message is a request from one object to another which causes an operation to be invoked on the called object. The object performs the operation and, optionally, returns a response.

Having data and methods together, as they are in objects, is referred to as *encapsulation.* Encapsulation means that the user of an object can know what operations are provided by an object, without knowing the details of how those operations are implemented. Encapsulation provides a level of protection against corruption of the object's data, as the data can only be manipulated by the methods provided by the object. This property is known as *information hiding.*

It is possible to place different objects into categories of objects or object types. Objects representing individual people, for example, could all be thought of as from the same category of objects. The object type of these objects would be Person. Object types are referred to as classes. "A *class* is an implementation of an object type. It specifies a data structure and the permissible operational methods that apply to each of its objects" [44]. Objects are *instances* of a particular class. Each instance of a class will share the same methods and data attributes

(although the value of those attributes may differ).

A class can be specialised into lower level sub-classes. For example the class Person may have subtypes of Civilian Person and Military Person. Military Person may have further types of Officer and Rating, and so on through a hierarchy of classes and sub-classes. A sub-class inherits the properties of its parent class through a process of *inheritance*. "Inheritance means properties defined for an object class are automatically defined for all of its subclasses"[33].[1] Inheritance is a very powerful feature of OO as it allows 'programming by difference', where only those parts of a class that are new need to be designed. This is essentially a form of software reuse.

Polymorphism is the ability to take on more than one form. In the context of OO, this means that an operation may be defined differently for different sub-classes. The appropriate implementation of the operation is chosen at run-time. The process of choosing the correct operation implementation is known as dynamic binding.

2.3.2 Object oriented analysis and design

Despite the availability of OO programming languages, it was not until the late 1980s, and early 1990s that OO design was given much, if any consideration.[2] The earliest work was started by Grady Booch, and by Shlaer and Mellor who published a book in 1988 [70]. Like Booch's early work, this represented OO systems as essentially entity-relationship models, ignoring the behavioural aspects of objects. Shlaer and Mellor later incorporated behaviour into their approach using standard state transition models. Coad and Yourdan [10] also developed a simple OO method which incorporated behavioural aspects. The simplicity of this method made it popular. Booch's work was also popular due to its direct support for structures in the C++ programming language, which was very popular at that time. This popularity lead to a huge explosion of interest in OO design methods, building upon this early work. So much so that by 1994 it was estimated that there were around 72 different methods available for developers of OO systems to use. Here just a few of the most influential methods are discussed.

The OMT method [67] introduced class models by incorporating operations into the entity-relationship type models. OMT was also influential by using data flow diagrams as a way of separating the processes of the system from the class diagrams. This concept of taking

[1] It is, however, possible for sub-classes to override inherited operations.

[2] It should be noted that the distinction between OO analysis and OO design is often unclear. Analysis is normally considered to be identifying and specifying the requirements. With many OO methods, these requirements are specified using specific design approaches, giving a *seamless transition* from analysis to design. For this reason the distinction between what is an OO analysis method, and what is a OO design method is often unhelpful. Here 'OO design' is also be used to refer to approaches involving OO analysis

different *views* of a system was a very important one for OO design. OMT took this further by emphasising the use of state transition diagrams to represent the lifecycle of objects, thus presenting another different view of the system.

Objectory [24] was a method developed in its OO form from an existing method used in the Swedish telecommunications industry by Jacobson et. al. The most interesting thing about this method is that the development of the design does not start with the class model, but from use cases, which describe the required functionality of the system. The classes are then derived from this use case model. Essentially what Objectory is proposing is yet another different view of the system. Objectory was also one of the first methods that attempted to suggest a development process for OO systems.

The Booch method [7] builds on Booch's early work to describe an approach which is very similar to OMT. Indeed Rumbaugh himself acknowledged that the similarities to OMT are more striking than the differences. One important concept in the Booch method is the use of Interaction diagrams as a way of tracing the execution of a scenario.

As was suggested earlier, the number of OO design methods available was unsustainable. In addition there was also a large amount of overlap between many of the methods. It was obvious that some kind of unification was required if OO was going to become successful on a commercial scale. One of the first attempts to combine the good parts of different methods together into one approach was with Fusion method [11], however this was felt by many to be poorly integrated. The breakthrough came when Rumbaugh and Booch, who had been working on combining OMT and the Booch method, were joined at Rational Software by Ivar Jacobson. What resulted from their collaboration was the Unified Modeling Language.

The Unified Modeling Language

The Unified Modeling Language (UML) [59] was accepted by the Object Management Group (OMG) in 1997 as the standard approach to OO modelling and has since become the de facto standard for most OO developers. In this section the way in which UML can be used to model OO systems is described in more detail. It should be clear that many parts of UML notation originate from the methods discussed previously.

UML represents designs using several views. A view is simply a subset of UML modeling constructs that represents one aspect of a system, such as the static view, or an interaction view. Here the design notations used for the different views are briefly explained.

The static view

The main constituents of this view are classes and their relationships such as associations and generalisations. “An association is a relationship among two or more specified classifiers that describes connections among their instances. Each instance of an association (link) is a tuple (an ordered list) of references to objects. The multiplicity attached to an association end declares how many objects may fill the position defined by the association end” [68]. It is the associations that turn a set of unconnected classes into a system. Generalisations are used to represent the hierarchical parent-class to sub-class inheritance relationships. The static view is represented in UML using a class diagram as shown in figure 2.19.

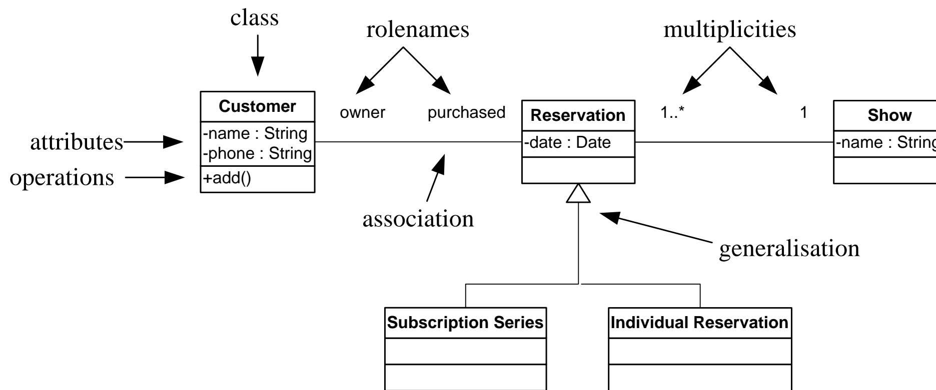


Figure 2.19: UML class diagram

During the execution of the system, instances of the classes represented in the class diagram (up to the number specified in the association ends) may be created and destroyed as required. It is possible to represent the objects and their relationships at a particular point in the lifetime of the system using an *object diagram*. This is essentially a special case of a class diagram showing instances instead of classes.

The interaction view

Objects must interact in order to implement behaviour. Understanding these interactions is therefore very important. The interaction view provides a view of the behaviour of a set of objects. This is done by modeling *collaborations*. A collaboration is a description of a set of cooperating objects assembled to carry out some purpose. “A collaboration describes the properties that an instance of a class has because it plays a particular role in a collaboration.

An object in a system may participate in more than one collaboration" [68]. UML provides two different diagrams for representing collaborations. Firstly the sequence diagram (figure 2.20) shows a sequence of messages for a particular collaboration arranged in a time sequence. The collaboration can also be represented as a collaboration diagram which represents the objects and links that are meaningful within an interaction. Figure 2.21 represents the same collaboration as in figure 2.20 in the form of a collaboration diagram.

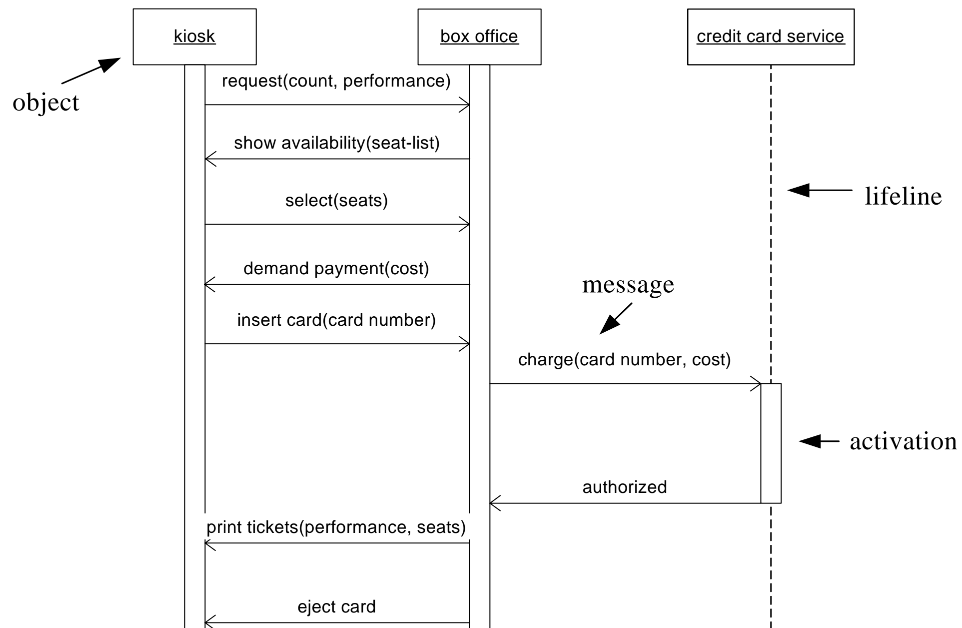


Figure 2.20: UML sequence diagram

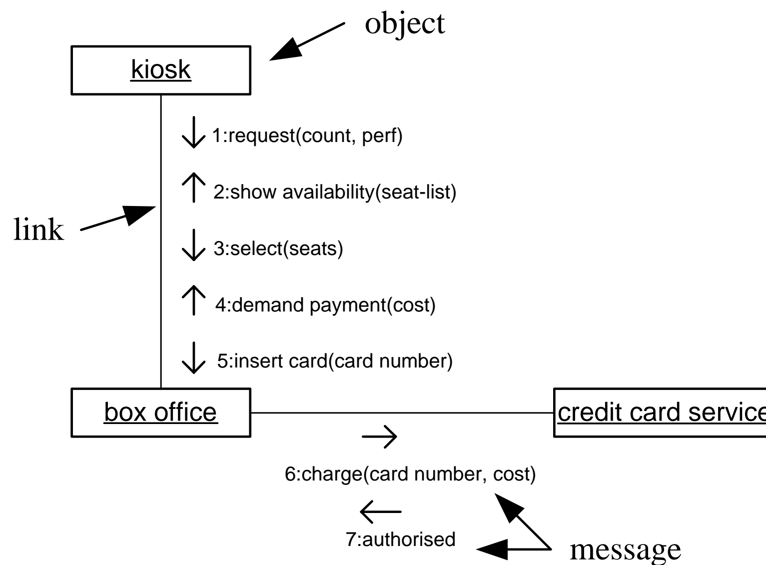


Figure 2.21: UML collaboration diagram

The two types of interaction diagram represent the same information, so which to use amounts

to a matter of personal preference. It is often easier to get a feeling for the timing of interactions using sequence diagrams. There is also the potential to annotate the sequence diagram with timing information, such as in [14]. The sequence diagram also makes it possible to represent the activation of a particular object, that is the time when that object is executing. The collaboration diagram is good if it is important to maintain some understanding of the static structure in the interaction view.

The state chart view

The state chart view is essentially another form of interaction view which focusses on a deep view of the behaviour of an individual object. Each object is treated as an isolated entity which communicates with the world by detecting and responding to events. The events could be, for example, the receipt of message calls, or the passing of time. As can be seen in figure 2.22, state charts in UML are essentially standard state machines which use states and transitions between those states to represent the lifecycle of objects of a class. Transitions between states may have a trigger event, which causes the transition, a guard condition, which is a boolean expression which must evaluate to true for the transition to occur, and an action, which is executed when the transition fires.

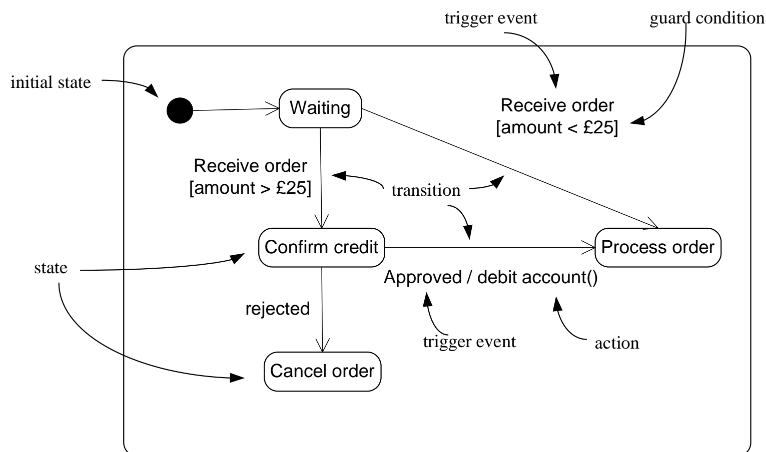


Figure 2.22: UML state chart diagram

State charts provide a relatively precise definition of object behaviour, and can even in some cases be used to directly generate code for the objects. The weakness is that the state chart provides a very narrow view which is separate from other objects, and it is therefore very difficult to get an understanding of the behaviour of the overall system. To understand the behaviour of the system it would be necessary to consider the combined effects of many state charts to-

gether. For this reason state charts are best used in conjunction with one of the interaction views discussed earlier.

The use case view

The use case view is used to model the functionality of the system as perceived by outside users, referred to as actors. A use case is a coherent unit of functionality expressed as a transaction between actors and the system. A use case is realised through a set of collaborations, therefore it is possible to represent different scenarios of a use case through interaction views. A use case diagram, as in figure 2.23 represents the use cases of the system, and links them to the actors that utilise that use case functionality.

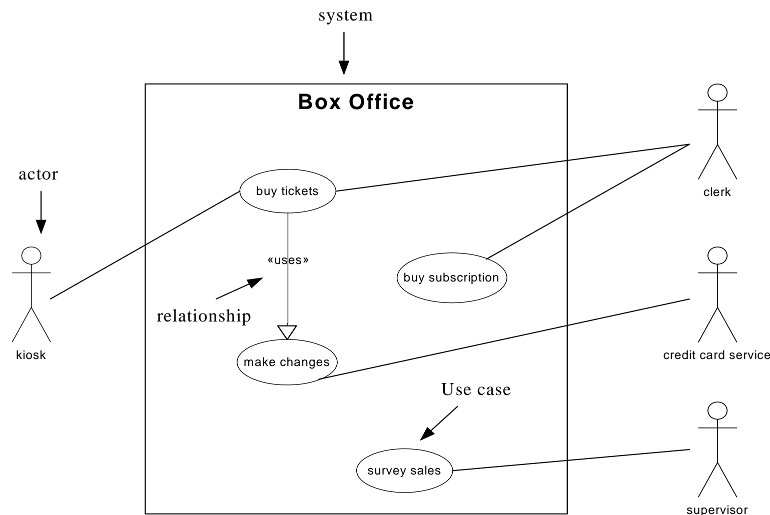


Figure 2.23: UML use case diagram

The use case diagram is essentially an informal high-level requirements capture for the system.

UML review

The views discussed above are the main ones necessary for describing any system however UML also provides other notations which may be used for providing different views of the system design. The activity diagram uses activity graphs to show a procedure or a work flow. This is the diagrammatical notation which has changed the most with the issuing of the new version of UML, UML 2.0 [61]. The notation includes connectors to allow the action flow to span multiple activity diagrams, there is also now a way of representing sub-activities and exception handling. There are many more changes, many cosmetic, which are not relevant to this thesis. It is worth

noting however that collaboration diagrams are now officially referred to as communication diagrams, however the old name persists. Within UML there is also the physical view which models implementation information. This provides the component diagram, which represents the implemented components and their interfaces, and the deployment diagram which can be used to represent computational resources and communication links.

It was identified as being of importance in the work presented in this thesis that the process developed fitted in as far as possible with existing software development approaches. UML is the most widely used and supported modeling approach for OO systems, and therefore it is UML models which will primarily be considered when looking at OO design models (although the process is applicable to any design methodology which provides the information). It is therefore necessary to acknowledge some of the weaknesses and limitations of UML. It should be noted however that there is no *perfect* modeling approach, and it is therefore crucial to understand the weaknesses of any approach such that they are considered appropriately in the development of the safety process.

As UML aims to support all concepts that arise in modern computer systems, it needs a large expressive power, which can tend towards a bloated and complicated language. This increases the potential for ambiguity and misuse. In [71] Simons and Graham identify 30 problems experienced by developers who used UML on real projects either in academia or in industry. By analysing these problems, the authors were able to identify four underlying causes of the failures identified in their survey. Firstly, they felt that being a universal notation, UML can have multiple interpretations. UML uses the same notation for modeling the analysis, the design and for documenting implementation. This means that a developer can interpret another developers diagram under a different set of assumptions. Secondly, the universal notation may also foster naive seamless development. For example a class diagram used at the analysis phase of development to represent concepts in the problem domain may become a concrete design. This 1:1 mapping was felt to rarely produce well structured systems. It was also noted that the use of classes and associations in the problem domain, as is the case with UML, can actively blind the developer to alternative practical structures.

The third underlying cause was identified as being that eclectic models fail to resolve competing design forces. As was discussed earlier, UML effectively has kept the best parts from a number of precursor methods and notations. This can mean that different model elements in a single UML diagram can often be in conflict because they originate from mutually exclusive design approaches. An example of this is in a UML sequence diagram which switches between dataflow

and method invocation perspectives. Models that were highly constrained in their original context may also lose their useful constraint when extra enhancements are added in UML. Finally, universal definitions of notational elements were felt to transfer poorly. Developers were seen to often take the interpretation of notational elements that they were most familiar with and retrofit those interpretations higher up the analysis process. The result of this is that it can lead to the imposition of implementation concerns too early on in the development process.

It should be noted that many of the things claimed above to lead to disadvantages in UML (such as its universal and eclectic nature) are the very same things that are also claimed elsewhere as advantages [68]. It is probably possible to conclude from this that UML used well can bring considerable advantages, whilst UML used badly can cause serious problems. So if UML (or indeed any modeling language) is to be used in developing safety critical systems, it is important that it can be constrained in such a way as to minimise potential ambiguity in the model.

There have been a number of attempts at constraining UML. The approach taken by Evans et al in [15] is to attempt to define a precise semantic model for UML diagrams. The motivation for their work is that they identify that a lack of precise semantics for UML is the main inhibitor to the reuse of specifications and designs. They identify two main problems. The first is that the generalisation relationship in UML only allows for a more specific element to be substituted for a more generic element. Although they acknowledge that this is useful, they note that there are other forms of incremental modification which are possible. They note that the semantics of the generalisation relationship are also unclear, particularly for packages.

Secondly, they identify that UML does not formally specify the relationship between UML diagrams. This means that, for example, the impact of modifications of a class in a class diagram on the state chart for that class or the collaboration diagrams describing the class's interaction with other classes is unclear. Reuse contracts ([73]) are identified as a way to deal with the evolution of OO class hierarchies and collaborating classes. Evans et al propose to solve the problems they identified by extending UML with a reuse contracts formalism, which gives precise semantics for the reuse and evolution of specification and designs in UML. This can be used to ensure consistency is maintained between different diagrams.

Another way of constraining UML is by representing constraints in UML through the use of the Object Constraint Language (OCL). The next section looks at OCL in some detail.

OCL

OCL is an expression language that enables one to describe constraints on OO models and other object modeling artifacts [82]. OCL has been accepted by the Object Management Group as part of UML and is described in the specification [60]. Constraints are simply restrictions upon values within the UML model. There are three standard stereotypes of OCL constraints: invariants, preconditions and postconditions. Invariants state a condition that must always be met by all instances of a class, type or interface. Invariants are expressions which evaluate to true if the invariant is met, and must be true all the time. Preconditions and postconditions are defined on operations. The preconditions must be true at the moment the operation is to be executed. Postconditions must be true at the moment the operation has just ended its execution. Unlike with invariants, pre and postconditions need only be true at a certain point in time and not at all times. Pre and postconditions express contracts upon the operations and are therefore discussed as part of chapter 4. The advantage of using constraints with UML is that it can improve the precision of the model, as constraints should not be able to be interpreted differently by different people. Constraints are also a way of enhancing the documentation of a model and capturing additional information. OCL allows an unambiguous communication of the modeler's intent, which is difficult to achieve with natural language. An example OCL expression is shown in figure 2.24.

```
context ClassName :: operationName(parameter1 : Type1) : ReturnType
pre: parameterOK : parameter1 > x
post: resultOK : result = y
```

Figure 2.24: An example OCL constraint

This section is not intended to be a tutorial on OCL, and therefore much of the detail has deliberately been omitted. This will be introduced as and when its use is required. This equally applies to the various extensions of OCL which are available.

2.3.3 Software contracts

Helm et al [21] introduced contracts as a way of explicitly specifying interactions amongst groups of objects. The idea of "design by contract" was introduced by Bertrand Meyer as a way of making OO software more reliable. In [46] he argues that reliability is even more important for OO systems. This is because reuse (for example through inheritance) is the cornerstone of OO and the potential consequences of incorrect behaviour are therefore even more serious as reusable

components may be in many different applications. It is argued that software elements should be considered as implementations meant to satisfy well-understood specifications, this can be achieved through contracts. Whenever a task in one software unit relies on a call to another unit to carry out a sub-task a contract exists between the two units. As in the real world, there are two major properties that characterise any contract. Firstly each party expects some benefits from the contract and is prepared to incur some obligations to obtain them. Secondly, these benefits and obligations are documented in some form of contract document. So if the execution of a certain task relies on a routine call to handle one of its subtasks, it is necessary to specify the relationship between the client (the caller) and the supplier (the called routine) as precisely as possible. This is done using assertions. Some assertions, called pre- and postconditions apply to individual routines, others, called invariants, constrain all routines of a given class. Meyer uses the Eiffel language [47] to represent pre- and post conditions as shown below.

Routine_name (argument declaration) is

    Header comment

require

    Precondition

do

    Routine body, i.e. instructions

ensure

    Postcondition

end

Preconditions express requirements that any call must satisfy if the operation is to execute correctly. The postcondition expresses properties that are ensured in return by the execution of the call. A precondition violation indicates a 'bug' in the client, the caller did not observe the conditions imposed on correct calls. A postcondition violation is a 'bug' in the supplier, the routine failed to deliver on its promises. Meyer explains that the presence of a precondition in a routine simply means that the client must guarantee that condition. It does not necessarily mean that the condition must be tested for before each call to that routine from a client. Another type of assertion that can be used is a class invariant. This is a property that applies to all instances of the class, transcending particular routines. In effect the invariant is added to the pre- and postcondition of every existing exported routine of the class and any subsequently added routines of the class.

In [25] Jezequel and Meyer look at the what they consider to be the reasons for the Ariane disaster discussed in chapter 2.1. As the report of the inquiry found, this was a reuse error, however the authors of this paper feel the real cause was the lack of any kind of precise specification associated with the reusable module. They propose that the *convert* function of the Internal Reference System horizontal bias module should have contained a require clause stating that the horizontal bias must be less than the maximum bias. They conclude that had the mission used a language and method supporting built-in assertions and design by contract then the crash would probably have been avoided. They do acknowledge however that it is always risky to draw such after-the-fact conclusions. They go on to state that the lesson to be learned from the event is that reuse without a contract is sheer folly.

Mitchell and McKim in [48] present benefits of using a design by contract approach gathered by talking to people who have used contracts. The first benefit they suggest is better designs. By this they mean that designs are more systematic, due to developers having to clearly and simply express obligations of client and supplier. There is also better control over the use of inheritance, which shall be looked at in more detail later. Contracts also provide a consistent meaning for exceptions and ensure they are used systematically. The second benefit is improved reliability. This comes from the developer having a better understanding of the code, and hence spotting faults more easily. Contracts can also help testing, which again increases reliability. Another benefit of contracts is better documentation, as contracts are part of the public view of a class's features and allow programmers to produce more precise specifications. Contracts provide easier debugging because bugs are easily pinpointed. Finally, contracts also provide support for reuse, again this will be examined later.

Mitchell and McKim also identify some potential costs and limitations of the use of contracts. Firstly they acknowledge that it takes time to write contracts, and although there is a downstream cost saving in terms of testing, documenting, reusing etc. it is often difficult to commit the extra time to upstream activities. Writing good contracts is also a skill that takes time and practice to acquire. There is also a danger that contracts may lead to a false sense of security, as contracts cannot express all the desirable properties of programs ("Does the presence of contracts ensure the safety of the system?" for example). Another potential limitation of a contracts approach suggested by the authors is that for some projects, an early release, even one with bugs, rather than the quality of the product is the most important goal. This should not, however, ever be the case for safety critical software!

Using contracts with OO systems

Contracts are ideally suited to use with OO systems. In [46] Meyer discusses how contracts can be used to provide a better understanding and control of inheritance. The system in figure 2.25 is taken as an example.

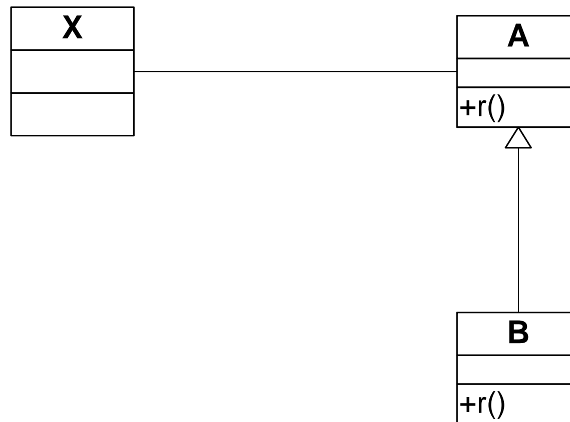


Figure 2.25: Redefinition of a routine under contract

In this example class A has defined a method r, which has a contract associated with it. Class B inherits r from class A but has redeclared it. Polymorphism allows A to become attached to instances of B. If X now makes a call to r then dynamic binding ensures that the redeclared version of r in B is called rather than A's original version. In effect A has subcontracted the task r to B. The problem here is that the author of X can only look at the contract for r in A. B could now violate its prime contractor's (A's) promises. B could do this in one of two ways. B could make the precondition stronger, risking some calls which are correct from X's view point not being handled properly. Or B could make the postcondition weaker, returning a result which is less favourable than what has been promised to X. This can therefore not be allowed. The reverse of these changes is permitted however. B can weaken the original precondition or strengthen the postcondition as both of these result in B exceeding the performance of the original contractor. The conclusion drawn by Meyer is that when using redeclaration in inheritance, the new version must remain compatible with the original specification, although it may improve on it. This can also be applied to class invariants, which must always be stronger than or equal to the invariants of each of its parents.

Several problems that could occur from making changes to reusable artifacts in a system are identified by Steyaert et al in [73]. The first problem arises when a parent class is changed to introduce a new method while one of the inheritors of that class had previously introduced

a method with the same name. Although this will not cause problems for the execution of the software as the parent class would be overridden by the inheritor's, the intention of the adaptation to the parent class is lost. Secondly, if a new abstract method is introduced to the parent class which is invoked by other methods of the parent class, then already existing inheritor classes which don't provide an implementation for this abstract class will be incomplete. If the new method were concrete, this could include extra method invocations of methods implemented by the inheritor that weren't invoked before. This is referred to as method capture and can result in erroneous behaviour as the inheritor did not take into account that its methods would be invoked by the parent. The inverse of this situation can arise when method invocations are omitted by a change to a parent class. The method is said to have become inconsistent with the method it used to invoke.

Reuse contracts are introduced by Steyaert et al as a way of managing such changes. These contracts are a set of either abstract or concrete method descriptions. These description also include a specialisation clause which list self sends (calls to other methods within the same class). There are also three operators introduced for reuse contracts, concretisation, refinement and extension along with their inverse operations abstraction, coarsening and cancelation. These operators allow the derivation of new reuse contracts on inheritors from the reuse contracts of the parent class. The derived reuse contract is labeled with how it has been derived based on the basic operators. When the base class changes it is then sufficient to check that the same operator can still apply to the new parent class, if it does then it is safe to conclude that no assumptions made by the reuser have been violated. This technique provides a useful way of documenting and managing change and reuse in a system and could potentially be applied to ensure safety properties are still assured in the presence of change.

Beugnard et al [5] describe how recent reports have found mixed results when components have been used and reused in mission-critical systems. They conclude that the solution is to be able to determine beforehand whether a given component can be used within a certain context. This should take the form of expressing what the component does without entering into the details of how. Contracts are proposed as a way of achieving this and they discuss how components can be made contract aware. It is proposed that contracts for components can be divided into four levels of increasingly negotiable properties. The first and most basic level is used to specify the component's interface and ensures that the client and server components can communicate. An example of this level of contract would be the IDL (Interface Definition Language) specification used in an object bus such as CORBA. This level of contract doesn't however define

the behaviour of the component, there is no indication of what the result of the execution of an operation might be. The design by contract that has been examined earlier in this chapter fits into the next level of contract as it defines an object's behaviour using assertions. These contracts assume that the services are atomic (either performed in their entirety or not performed at all) or executed as transactions (a defined sequence of atomic services). The higher layer of contracts deals with the global behaviour of components in terms of synchronisation between method calls. This is particularly important were one server component has many clients. The contract will guarantee to a client that the service will be executed correctly, whatever other clients request. Beugnard et al propose achieving this through the addition of synchronisation policies to a contract. The fourth level of contract is the quality-of-service contract which can quantify the expected behaviour or offer the means to negotiate these values. The difficulty with using contracts such as these is that they rely on third parties.

Review of contracts

It has been seen how contracts can provide a very powerful and effective way of constraining the behaviour of software. This is particularly true for OO software. As Meyer noted, whenever a task in one software unit relies on a call to another unit to carry out a sub-task, a contract exists between the two units. This is exactly the situation that exists with OO systems where it is object collaborations which realise the functionality. It is important to note that the presence of contracts alone is not enough. The constraints defined in the contract must be correct, and must be correctly implemented by the software. It has also been seen how contracts can provide support for reuse and inheritance. It is important that any constraints on the system design support such features, as they are beneficial features of OO.

2.3.4 Analysing OO designs

The review of OO has so far looked at the generation of OO designs, and the use of contracts as a way of constraining those designs. For a safety critical system, as was seen earlier in this review, it is important that analysis of the design is used to define the constraints such that they ensure that hazards do not occur. In this section analysis approaches for OO systems are reviewed. This will help in the definition of a safety analysis method in this thesis.

The first techniques that shall be looked at are those suggested by Nowicki and Gorski. They introduce three methods for OO safety analysis in [54]. The first method, which is described in

detail in [18], is based around a safety analysis of the system mission. As such, this does not take into account random failures of any of the objects that make up the system. The technique involves enriching the high-level system model with a hazard model. This model consists of explicit safe and unsafe states and the transitions between these states. Critical objects in the system (objects to which the hazard refers) are identified and merged with the hazard model. Reachability analysis can then be used to check if the hazardous state is reachable within the mission constraints. Any hazard scenarios identified can be used to reconstruct the system to ensure safety. Although the idea of identifying if objects in the system can reach hazardous states is a sound one, the method as proposed in this paper has shortcomings. The identification of critical objects is not dealt with adequately. The procedure outlined states that critical objects are those objects the hazard definition refers to. There is an assumption here that the hazard definition has correctly identified all critical objects, but no guidance on forming the hazard definition is given. It is also necessary to produce the hazard model, again little guidance on how this is to be produced is supplied and it relies on the hazard definition. The final stage is to change the system design to prevent the hazard. Although in the example given in the paper it is clear why the particular changes were made, the analysis does not explicitly guide the analyst towards the necessary changes and it appears to be very much down to the skill and judgement of the designer.

The second method they propose is detailed in [17]. This is a method to analyse the impact of errors on the safety of the system. The first method assumes that the system and the environment are reliable in the sense that they behave as specified. This method accepts that this assumption isn't always valid, as objects are exposed to random failures and the environment can violate assumptions made about it. This method provides a set of templates of faulty behaviour which are deviations from the normal behaviour of the object. Fault initiator objects are identified, these are objects which can influence other objects by sending events and/or by being in some state or assigning a value to the variable that is sensed by another object. Dynamic models of the fault initiator objects are developed and the templates of faulty behaviour used to produce a model of the 'unreliable object'. The templates of faulty behaviour are developed by considering possible faults in transitions. As has already been seen, transitions can be made up of three elements, an event, a condition and an action. The authors propose that by considering possible faults for each of these transition elements a list of rules for constructing improper transitions is derived. These rules are applied to the dynamic model of the initiator object. Reachability analysis is then performed to see if the hazard can occur for this unreliable object. The next stage is to identify how the faults of the object can propagate through the

object model. Blocking objects are then introduced to stop the fault propagating. The hazard reachability analysis is then re-performed to check the effectiveness of the blocking object. Again there are shortcomings with this method. It is not clear how to identify which object may be a fault initiator. The paper identifies this by experience of which object is likely to behave in a faulty manner. This does not seem an adequate approach for general application to complex systems. The introduction of blocking objects as a way of stopping a fault propagating around the system, although sound in theory and in the simple example used in the paper, could be highly problematic for larger systems with many faults where appropriate blocking objects may not be so evident and a huge number of extra objects may need to be introduced.

Whereas the first two methods were concerned with verifying the system design meets its safety constraints, the final method which is introduced by Nowicki and Gorski aims to strengthen the safety guarantees of the system. This is done by enriching the system with a device they call a safety monitor. The only concern of this is safety. If a potentially hazardous sequence of events is developing in the system then the safety monitor will execute corrective action before the hazardous behaviour can manifest itself as an accident. The safety monitor is formed using a model of the critical objects in the system. If the safety monitor is stimulated with the same events as the system then it will detect if a hazardous state will result in the actual system.

Lano, Clark and Androutsopoulos in [37] examine how safety analysis techniques, predominantly HAZOP, can be adapted to OO systems, particularly UML. Firstly they consider state transition diagrams, with the attributes event and action of transitions as the basic elements of analysis. The guidelines for hazard analysis of state charts given in Def-Stan 00-58 [50] give the following interpretation of guidewords for event:

No Event does not happen

As well as Another event takes place as well

Other than An unexpected event occurs instead of the expected event

The authors consider this interpretation to be inadequate and instead suggest the following revised guideword interpretation:

No Event not received by control system: either it occurs but is not transmitted to the controller because of sensor or other failure, or it does not occur even though expected

As well as Another event is detected by the control system as well as the intended event

Other than An unexpected event is detected instead of the expected event

Similarly for the action element, a revised from:

No No action takes place

As well as Additional (unwanted) action takes place

Part of An incomplete action is performed

Other than An incorrect action takes place

To the following interpretation:

No No action is produced by the controller, or this action is not transmitted to / carried out by actuators

As well as Additional (unwanted) actions are generated / performed

Part of An incomplete action is generated / performed

Other than An incorrect action is generated / performed

Class diagrams are also considered in 00-58, however it is noted that the guideword interpretations suggested tend towards consideration of design flaws of the diagram e.g. "there is a required relationship that is not shown on the diagram" rather than deviations from the design intent of the system being described. Therefore the following guideword interpretations for relationships in a class diagram are suggested:

No No information about this relationship between two objects is recorded by the control system, even though the relation is true in the real world; or the relation does not hold between two real-world objects when it is expected to

More / less The number of objects in the relationship with another does not obey the cardinality restrictions expected. This may either be a feature of the real world or erroneous data held by the control system

Part of Some semantic constraints of the relation given in the diagram hold, but others do not (either in the real world or in the control system data)

Other than The specified relation does not occur between some objects, another unintended relation is present instead between these objects

Similarly, guideword interpretations for the classes themselves and attributes of those classes are suggested. The same approach is applied to sequence diagrams where interpretations for guidewords for objects and messages within these diagrams are given.

This paper is very useful for suggesting how HAZOP may be adapted to OO systems. If this technique were to be applied to every class, attribute, state transition, and interaction in the system design of even a fairly small system it would demand considerable time and effort. No guidance is given on how the technique can be applied in a more focused way. This paper also doesn't suggest how the results of this analysis are used to ensure safety.

In [78] Tsuchiya et al use fault trees to derive safety requirements from requirements specifications for OO designs. Their approach focuses exclusively on statecharts. A high level fault tree is constructed and a safety requirement is derived for each basic event in the fault tree relating to the software. The example given in the paper is that the fault tree identifies 'door is open' and 'box is moving' as basic events leading to the 'passenger is injured' top event. The safety requirement is therefore derived that the door should not be in 'open' state at the same time as box is in either 'up' or 'down' state. The authors verify the requirement is met in the design using tables of possible states for the objects in the system. Although the idea of breaking down higher level requirements into lower-level safety requirements on the software using fault trees seems a good one, it is felt that the approach in this paper is slightly odd. The object state charts are a very low-level design technique. The analysis has gone from a system hazard (actually more correctly an accident in this example) to states in individual objects in just two layers of fault tree decomposition. It seems that generalising this approach to complex designs would be impossible. It seems that the high level fault tree is being applied at the wrong level of design abstraction. The author feels that fault trees would be more appropriately applied to higher level views of the design, before failures are broken down to a more detailed level through further analysis.

In [87], Wong explores a way of deriving "safety verification conditions" for OO designs from system hazards. This is done by using interaction diagrams to identify critical software components (objects) in the design. Once these have been identified, detailed design criteria necessary for safety are generated using the Verified Tree Method, which is essentially an extension of fault trees. These design criteria are then used in modifying the design of the critical components. An important aspect of this paper is the desire to express the design criteria in UML such that they are easier to communicate to the software developers.

Allenby and Kelly in [1] propose a method for using use case scenarios in order to perform hazard

analysis. Their approach involves applying HAZOP to use case scenarios in order to identify hazards. Where necessary additional use cases are added in order to deal with these hazards. A similar approach is also proposed in [27]. These are useful and interesting approaches to hazard analysis which fit in well with an OO development process. Use cases however take an external view of the system, and the internal design is not considered. For this reason there is nothing that is especially object oriented about the approach, and indeed it could just as successfully be applied to any standard system.

A paper produced by Artisan software [75] explores the safety features within standard UML. This looks at the high-level safety features of different UML views. For use case diagrams, the important safety issue is identified as being able to understand which actors have access to which service. It is noted that it is information which is not on the diagram that can often be the safety feature of the system. For example it might be important that a particular actor cannot access a particular service. The safety features of interaction diagrams are said to be "by exploring the detailed interactions of 'how' the system service (use case) is delivered over multiple scenarios, the safety engineer has a far more detailed view of the internal workings of the evolving system" [75]. With respect to class diagrams it is noted that for some safety systems where dynamic allocation of software objects is not permitted due to the non-determinism of memory usage that can result, extra details are required to ensure a suitable 'collection mechanism' has been defined to support the 'many' relationship between object classes. For state diagrams it is noted that safety engineers should concentrate on the transitions between states and that guard conditions can be used to inhibit transitions between states even when the event associated with the transition has 'fired'. This paper explains how safety requirements for the system may be represented using UML notation. This seems to involve simply using the notation in the standard way it would be used for representing any system requirements. The paper seems to miss the fact that to ensure the safety of the system it is not sufficient to just consider the representation of requirements, but actually deriving the correct safety requirements in the first place, and then being able to check that a system design will meet all those requirements is just as important and challenging, if not more so.

2.3.5 Representing requirements

Previously in this section, the use of OCL as a way of specifying constraints within UML design was discussed. The purpose of OCL in UML models is to provide a way of writing unambiguous constraints, without having to use traditional formal languages. Avoiding formal languages

ensures that the constraints being represented are easily understood by the system developers and the constraints can form part of the design, rather than being viewed separately. There are alternative ways of representing constraints in OO designs, some of which are discussed here.

Since most safety critical systems are real-time in nature, many of the constraints necessary to ensure the safety of the system will be constraints upon timing. In [14], Douglass provides a detailed analysis of modeling real time properties effectively using UML. Douglass proposes many patterns that can be used to capture important real-time features such as component synchronisation, transactions and watchdogs. He also makes use of non-standard UML notations such as timing diagrams and task diagrams. The design of systems is looked at on three levels: the architectural design, which deals with processors, components and tasks, the mechanical design, which deals with groups of collaborating classes, and the detailed design, which focuses on the class level. The way in which key properties at each level may be specified is discussed.

In [28], the author describes how the UML extension mechanisms can be used to include safety requirements in a UML model. The author suggest a "safety checklist" of stereotypes which he suggests should be used when developing safety-critical systems with UML. The stereotypes are defined on different UML elements. The following stereotypes are defined for links:

- risk
- crash failure semantics
- value failure semantics
- guarantees

The following stereotypes are defined for subsystems:

- error handling
- containment
- safe behaviour
- safe communication links
- safe dependency

In addition there is a critical stereotype for objects and a redundancy stereotype for components.

The author states the goal of the approach to be "to enable developers without a background in safety to make use of safety engineering knowledge encapsulated in a widely used design

notation." Indeed safety engineering knowledge expressed in this form would be accessible to developers. However, there is an assumption that that safety engineering knowledge (i.e. safety requirements) is known, and is available in a form that can be expressed using the stereotypes. The elicitation of such knowledge is outside the scope of the paper.

2.3.6 Verification of OO

As was mentioned previously, this thesis does not set out to define or recommend verification techniques. However it was seen in this review that part of the safety process for any system is verification that the system meets its safety requirements. It is therefore important to have an understanding of the verification techniques available, so that safety requirements may be specified in a manner which facilitates their verification. Both dynamic analysis (testing) and static analysis techniques shall be considered.

There are certain of the characteristics of OO systems that were discussed earlier, such as encapsulation, information hiding and inheritance, which make traditional verification techniques insufficient for OO programs. There are also aspects of OO systems which facilitate testing. Firstly encapsulation is considered. As observed by Barbey [3], the notion of encapsulation can be very helpful for testing as a class can be tested in isolation from the rest of the system, i.e. the context in which the test is made does not affect the testing of the class. Another advantage that encapsulation brings for testing is that the dependencies between objects tend to be explicit and obvious.

Encapsulation can also bring problems however. The attributes represent the *state* of the object, which is changed by operations altering the value of the attributes. What this means is that the behaviour of the object is dependent on the object's state at the time the operation is called. This has an effect on what is defined as a *unit* in unit testing. Traditionally the unit would be a subprogram or subroutine. Because of encapsulation, the state of the object must be considered when testing a method. A method would not then be a suitable unit for testing, as its behaviour depends on the object state, which is in turn dependent on other methods. Therefore the smallest basic unit for the unit test has to be the object or the class. Encapsulation can also have an impact on integration testing. Integration testing combines many tested modules into subsystems which are incrementally tested. In procedural programming, the modules are subprograms, but in OO systems they are objects instead. It is necessary to consider if objects can be integrated into systems in the same way that subprograms can be. If they can't then the impact on the integration test strategy needs to be considered.

Information hiding can also make life difficult for testers. In order to test an operation, one may want to check the object's state before and after invocation. Information hiding however means that the internal state of the object may be hidden from the tester. State reporting methods (e.g. getAttX()) can be used to inspect the internal state of an object by returning attribute values, however there may not be a state reporting method for each internal data item. Even if there is, these methods themselves may not have been validated and therefore cannot be trusted. A strategy for testing these state reporting methods is therefore needed. The unit for abstraction in an OO system is the object, and this can essentially be treated as a black-box. This means that we are told what functionality an object can provide, but not told information about how the functionality is implemented (the object is essentially defined by its interface). This black box view is acceptable from the point of view of the client object who only needs visibility of the interface, however it makes life difficult for the tester. The information that has been abstracted by the object is required if it is to be satisfactorily tested.

Inheritance must also be considered when testing OO software. When testing inherited classes there are two extreme possibilities. The first is to test the inherited class as a flattened class. This is a representation of the class which includes all attributes and operations inherited from classes higher in the hierarchy, as well as those added or changed by the inherited class. This effectively means that all the properties of the base classes are being completely retested for the inherited class. Although this approach is sound and thorough, it means that no benefit is gained, from a testing perspective, through the use of the inheritance mechanism. This is a long way from a *'testing by difference'* approach, where only the aspects that have changed need to be tested.[3]

The second extreme case is to assume that if the base class has been adequately tested, then its properties in a derived class do not need to be retested. There are a number of reasons why this approach is unsound. The first is that overriding allows a different subclass implementation of a function. This could be in the form of a different algorithm, different functionality, or both. The test suite that was used to test the overridden method in the base class will almost certainly not be suitable to test the overriding method. It is also necessary to consider the effect of functions in the derived class on attributes of the base class and vice-versa. Neither of these extreme positions is acceptable. The best solution is certainly somewhere in between the two extremes.

[3]There is debate as to whether reuse of testing is achievable at all, such as [62], however it is felt that this is still a noble and realistic aim

Adequacy of existing testing methods

The discussions above indicate that existing approaches to testing will not be sufficient for OO programs. Weyuker[85] has proposed an axiomatic theory of test data adequacy. In [62], Perry and Kaiser apply these axioms to OO programs and conclude that adequacy can only be achieved if the following conditions are applied. When a new subclass is added, or an existing subclass is modified the inherited methods from each of the super-classes must be retested even if they were already thoroughly tested. Class-level testing is required even if every method in a class is individually tested. Retesting components in most contexts of reuse is required. These observations seem to fit with those made previously in this section. In her PhD thesis [32], Kim also looks at the adequacy of traditional testing techniques for OO systems, and attempts to identify why they are insufficient. She identifies that traditional test adequacy criteria (program mutation methods) are not reliable for assessing the adequacy of OO software testing. The choice of mutation operations for OO programs should be extended to deal with new characteristics of OO. The additional mutant operators for an OO language (in this this case Java) are generated using a HAZOP analysis of Java language constructs. These constructs are claimed to be adequate for successful use of mutation testing on programs written using Java 1.0. In the rest of this section some approaches to addressing the issues raised above in verifying OO systems are briefly examined.

Base class unit testing

The purpose of unit testing is to assure that the individual parts of the complex system work correctly in isolation, before their eventual integration. Our interest is in testing a class for its correctness and its completeness. Correctness means that the class delivers the service it has promised to perform, and responds acceptably in the face of unexpected conditions [4]. Completeness involves checking that the class has all the necessary functionality, that the function is available at the public interface, and that each method completely executes its specified responsibility [4]. If a unit larger than a class were chosen, then completeness would be harder to achieve as it is easier to get coverage of all possible paths at the class level.

Correctness of a class depends on whether the data attributes are correctly representing the intended state of an object, and whether the class' functions are correctly manipulating that representation. Perry and Keiser's Anticomposition Axiom [62], indicates that adequately testing the individual methods of a class does not ensure that the class has been adequately tested.

This is because the state of the object during testing must also be considered. In order to do this it is necessary to test each of the data attributes separately. To do this, a class can be viewed as a composition of a set of *slices*. A *slice* of a class can be defined as a quantum of a class with only a single data member and a set of member functions such that each member function can manipulate the values associated with this data member [4]. Figure 2.26 shows how a class can be split into slices. It is assumed that the data attributes are independent.

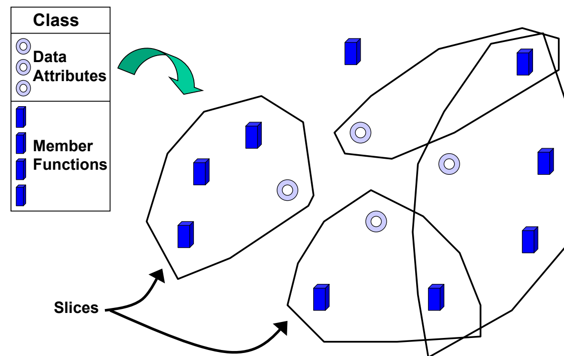


Figure 2.26: Splitting a Class into Slices

Bashir and Goel assert that testing all sequences of the methods in a slice is equivalent to testing the class for this slice or this data member. They claim that if all slices of a class are tested in this way, it can be concluded that all member functions are correctly manipulating an object of that type.

Derived class unit testing

In the testing of derived classes, a slice based approach can again be used. It is important when forming the slices however, that as well as considering functions within the class itself, member functions of the base class are also considered. It is possible to think of a matrix containing four quadrants as illustrated in figure 2.27

Quadrant 1 captures those derived class member functions which could impact the data members of the base class. Quadrant 2 is the base class testing as discussed in the previous section. Quadrant 3 deals with functions which have been inherited from the base class, which refer to data in the derived class. These base class functions must be identified and included in the slice for the data member of the derived class. Quadrant 4 is in effect doing the same as for the base class, but for the inherited class. If it had been assumed that the properties of a base class do not need to be retested in a derived class then it is only quadrants 2 and 4 that would

	Base class member fns	Derived class member fns
Base class data members	2	1
Derived class data members	3	4

Figure 2.27: Matrix structure for testing derived classes

have been tested for. The relationships in quadrants 1 and 3 would have been missed. This approach is a good way of ensuring that all the necessary dependencies are tested for.

In [19], Harrold et al. describe a method for incrementally testing inherited classes. This involves the use of a testing history which stores and controls the execution of the test cases for each class. For each class the test case is formed by combining existing tests from its parent classes, with new ones developed specifically for the new class. In essence this is very similar to the approach described above, however the process for deciding which tests to reuse is different.

Integration testing

The complexity and interdependencies of an OO program makes testing of such programs difficult [35]. Traditionally, integration testing would focus on ensuring that the interfaces between individual units are correct. For OO programs it is necessary to consider the state associated with each of the units. Kung et. al. [35] suggest the use of an Object Relational Diagram (ORD) to capture the different relationships (inheritance, aggregation and association) that exist between classes. As before, it is necessary to construct slices for testing, as the state associated with the individual objects is important. The order in which the tests are undertaken also needs to be considered, with the aim being to find an order to test the classes such that the effort required to construct the test stubs is minimum. In [35] the authors describes a test order identification algorithm which will compute an optimal test order for unit and integration testing of OO programs.

The features of dynamic binding and polymorphism also make integration testing of OO systems more complex. The procedure called cannot be determined statically as it will depend on the run-time determination of the class of which the variable is an instance. This can result in many possible combinations of classes at run time, and these relationships can be hard to

capture. This is also a problem for static analysis and this shall therefore be considered in more detail shortly

Object state testing

An alternative approach to testing OO software is to use state-based testing. In certain cases it may be easier to detect errors using this approach than with more conventional functional and structural testing as discussed above. Kung proposes a method for object state testing in [34]. In [79] an approach to integrating state-based testing with more conventional functional and structural testing is described.

Static code analysis

Static Code Analysis (SCA) has been proven to be a powerful software verification technique, which provides the necessary rigour for safety-related software. Indeed many of the standards discussed earlier such as Def Stan 00-55[51] and IEC 61508[23] mandate its use for safety-critical applications. As Sampson points out however [69], it also has a reputation for being costly and labour-intensive. He believes that this is mainly because it is generally felt that it is necessary to demonstrate that *all* the software requirements have been correctly implemented. If SCA is used only to analyse the *safety* requirements, then it becomes much more viable. As was seen seen to be the case for testing, the use of OO can bring particular difficulties when using SCA. This is again due to the presence of polymorphism, and particularly overridden functions (that is ones which are re-declared in the sub-class).

All overridden functions are essentially hidden branching statements in the program which are difficult to trace. This is because the actual target for a function call isn't known until run-time, and therefore, statically, there are many possible options for the target function. These options for polymorphic calls may be buried in several different classes scattered throughout the system. This makes it difficult to identify and collect the different options for all of the possible branches. It may even mean that one is unaware that a function is overloaded at all. There are SCA tools available, such as those discussed in [86], which will generate calling trees to assist in the analysis. However there can still be a huge combinatorial explosion in the number of possible paths, thus impinging on the ability to perform SCA successfully on OO software.

An interesting approach to avoiding this combinatorial explosion [12] is to take advantage of

the Design-By-Contract (DBC) approach which was discussed earlier. The way in which DBC helps with the dynamic binding problem is that the caller of the method need refer only to the precondition of the nominal target in order to be guaranteed the postcondition by the actual target. In this case the actual target does not need to be known by the caller and we no longer have the explosion of branches. This is reliant however on the contract of the nominal target being correctly linked to the contract of the actual target which is actually satisfied. In order to ensure this, Liskov's substitution principle (LSP) [42] must be applied. LSP requires that the contract for a method in the subclass must conform with the contract in the superclass. Again, to put this in simple terms, the new contract (for the actual target) may assume no more than the old (for the virtual target), and it must promise no less. This is often paraphrased as "weakening the precondition and/or strengthening the postcondition". Crocker goes on to discuss in [12] how contracts can be formally verified.

Review of OO verification

The use of contracts as a way of specifying constraints on the system design has been shown to have advantages for static code analysis in dealing with dynamic binding. It should also simplify class unit testing as it provides clarity on what it is that is being tested for, this clarity should also help to make SCA more viable. This suggests that the safety contracts approach to OO safety being proposed in this thesis could be beneficial when it comes to verification of the software. Contracts could potentially cause complications for integration testing, as it is necessary to ensure that any caller meets the preconditions, however this will essentially just mean that there are additional requirements to check. In this section the focus has been on class testing, as this is by far the most common approach. Some other approaches such as [53] use cluster testing, where clusters of classes are the smallest tested unit, which is perceived to be cheaper, both in time required to write code to support the class, and in time writing test plans.

2.4 Thesis Contribution

Based on the results of the survey performed in this chapter it is possible to identify the contribution that this thesis makes. Below are some problem statements which are addressed by this thesis.

- Ensuring the safety of an OO software system requires that the contribution of the software to system level hazards be identified and mitigated.

- There exist a number of techniques for analysing OO software designs, however, there is a lack of a coherent process.

- As part of any such process, it is necessary that safety requirements may be generated in a way that supports the OO paradigm.

- There exists little guidance on how a defensible safety argument may be produced for an OO system based on the use of a combination of techniques.

- It is essential that safety is considered during all stages of the development lifecycle of the software.

Based on these requirements, the proposition from chapter 1,

> Through the development of safety contracts, it is possible to establish a systematic, thorough and scalable process to effectively support the use of an OO approach when developing software for safety critical systems.

is supported in this thesis through:

1. The development of a hazard-driven, product based safety analysis process for OO software.

2. The integration of safety requirement elicitation as part of the process.

3. The development of safety argument patterns which support the use of this approach.

In chapter 7, the success with which this thesis addresses the proposition is evaluated.

2.5 Conclusions

This literature survey firstly looked at the safety lifecycle, and identified current best practice for developing safety critical systems, and particularly the approach and techniques that are available for developing safe software. This part of the review identified what needs to be achieved for an OO system if it is to be accepted for use in a safety critical role. The review then went on to look at the OO paradigm and explored the way in which OO systems may be

designed. The way in which these designs may be analysed and verified was then discussed. Finally the way in which safety arguments can be constructed for software systems was explored. The results of the literature survey have been used in identifying the contributions that are made by this thesis.

Chapter 3

Performing Safety Analysis of OO Systems

3.1 Introduction

The literature survey in chapter 2 explored two separate areas. Firstly the safety analysis process for safety critical software systems was discussed. Then the unique characteristics of the OO paradigm were explored. The aim of this thesis is in essence to propose a method to allow an OO approach to be safely used for safety critical systems by building on existing practices and results. Chapter 2 showed how the first part of the safety process which explicitly considers software is the PSSA stage, where the design is analysed, and derived safety requirements are produced. It is the PSSA stage which the author contends to be a crucial area that must be addressed for OO software. It is felt that although certain aspects of analysis and specification are dealt with in great detail (such as Gorski and Nowicki's state chart analysis [54] or Jurjen's UML stereotypes [28]), the current literature does not adequately consider the nature of the derived safety requirements that need to be generated for OO software. The analysis that is necessary to perform will be determined by the nature of the safety requirements. Producing safety requirements which are in the correct form is seen as being crucial to successfully providing an integrated and verifiable approach to assuring the safety of the system. The nature of derived safety requirements for OO systems is not properly addressed elsewhere and is a major contribution of this thesis.

Chapter 2 looked at many techniques which are available for performing PSSA on software,

including some techniques that specifically analysed OO designs. This chapter will propose an approach for carrying out the analysis, however as was discussed above, in order to understand the best approach for PSSA on OO software, the nature of the safety requirements which the analysis will generate must first be understood.

3.2 A Framework for Analysis

As discussed in chapter 2, one of the principal aims of PSSA of software is deriving a set of software safety requirements which are sufficient to ensure that the behaviour of the software will not contribute to the identified system hazards, and thus ensure that the system is safe. The approach can be summarised as shown in figure 3.1.

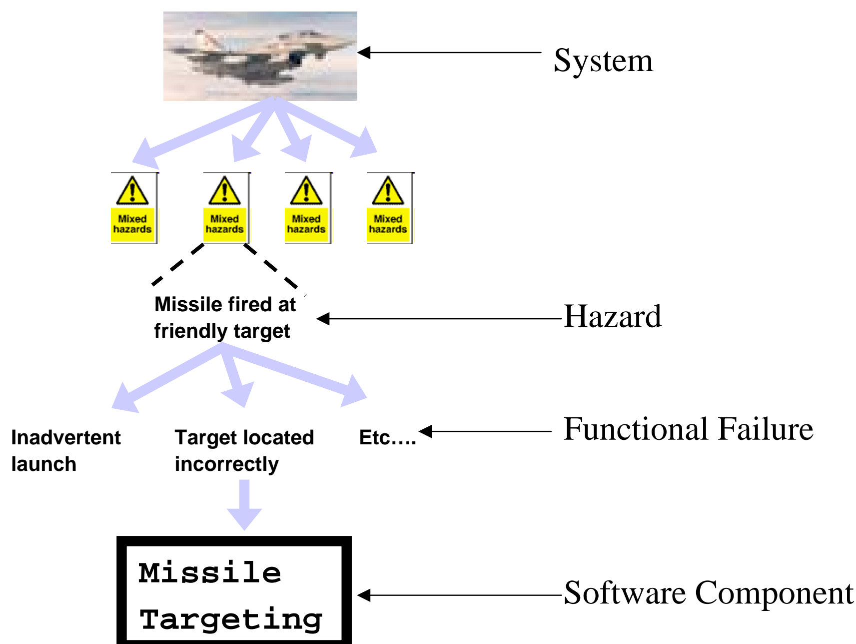


Figure 3.1: Deriving safety requirements for a software system

The safety process starts at the level of the system. The hazards associated with the system under consideration are identified. Through performing the hazard analysis, it is hoped to understand how these hazards may be brought about, so that they may be prevented from occurring or otherwise controlled. The next stage of analysis is therefore to identify the failures which may occur at the sub-system level to contribute to the hazards of interest. Chapter 2 discusses the techniques available to perform such analysis. These higher level PSSA techniques (such as FTA) are applicable to any software system, including one developed using OO.

Once the failures which may contribute to the hazards have been identified, it is necessary to

specify derived safety requirements for the software which prevent those failures occurring or detect and mitigate them. It is at this point in the analysis that the existing techniques become more difficult to apply to an OO system. With a *traditional* (i.e. functionally decomposed) software system the functions will be allocated such that there will generally be many black box sub-systems responsible for different functions. The safety requirements can be assigned to the relevant sub-system and the software implemented to meet these requirements.

For example in figure 3.1, the functional failure of interest is 'Target located incorrectly'. Due to the functional decomposition of the software components in the system, it is fairly easy to identify which part of the system may contribute to that failure (i.e. there will be part of the system which has responsibility for that function). So safety requirements could be derived for the missile targeting component.

If we now consider an OO system, as was discussed in chapter 2, we know that in such systems the software is not directly decomposed according to the system functions, but into a set of objects. There will not be a 'missile firing' object. The missile firing function is realised by a number of collaborating objects passing messages between themselves. In this case how can the safety requirement be assigned? No one object has responsibility for a particular function. As illustrated in figure 3.2, there is no longer a direct relationship between a functional failure, and one particular part of the software system. In addition, objects may be involved in many different system functions as part of separate object collaborations. In addition, the basic unit of any OO system is not the function but the object. To support the use of an OO approach, and facilitate reuse, it is desirable that safety requirements can be derived for objects. The safety requirements obviously need to be broken down further in some way. This raises the question of how this may be achieved.

In order to answer this question it will be necessary to better understand the way in which faulty behaviours, and therefore hazards, manifest themselves in an OO system. Firstly, the way in which functionality is realised in an OO system is examined in more detail.

It was previously mentioned that functionality is achieved in an OO system due to a collaboration between interacting objects. The way in which this occurs is illustrated in figure 3.3. This shows four different objects, O1 to O4, with a sequence of interactions occurring between them, i1 to i3. Within this thesis, an interaction is defined as any communication which occurs between objects in order to perform a function. Each of the objects can be considered to have a set of states associated with it. The object can therefore be represented using the statechart notation described in chapter 2. When an object interacts with another object, this interaction

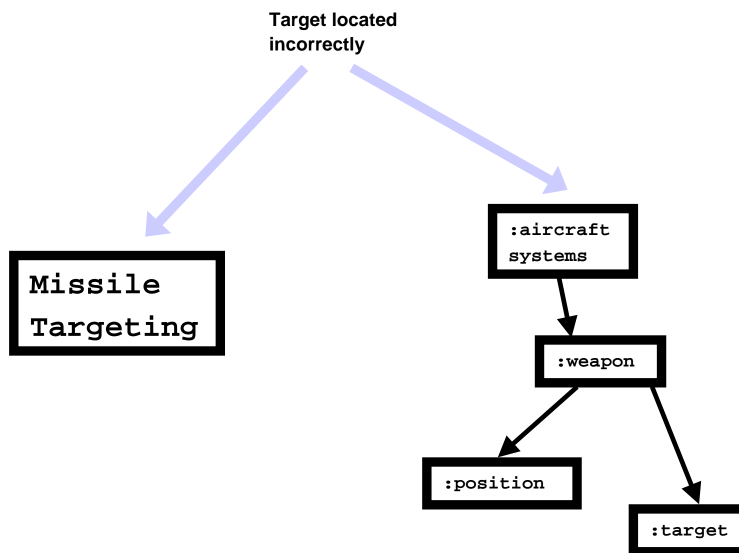


Figure 3.2: Identifying software contributions for functional (left) and OO (right) systems

can trigger that other object to change its state. For example in figure 3.3, the transition of object O2 from state A to state B is of the form: i1[c]/i2. So if O2 is in its initial state A, then the interaction, i1, sent from object O1, will cause the transition to state B if condition c is met. As well as changing the state of O2, this transition will also result in the event i2, which is an interaction with object O3. In the example in figure 3.3 this can cause a further transition to occur in O3, and so on. It is through such interaction sequences that an OO system realises its required functionality.

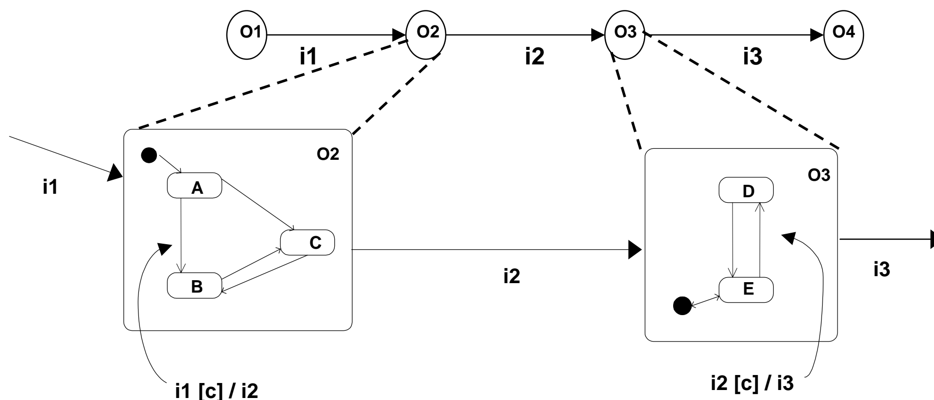


Figure 3.3: Sequence of interactions occurring in an OO system

It is the interactions between the objects which are crucial to the safety of OO software. These interactions must be constrained such that the software's behaviour will not contribute to hazards. The survey in chapter 2 identified that contracts provide an excellent way of constraining such interactions. Chapter 4 will look in more detail at the specification and use of contracts for

this purpose. The analysis carried out on the OO design at the PSSA stage must provide the necessary information for forming safety requirements in the form of contracts. This chapter explores an analysis approach for gaining such information.

3.3 Developing an analysis process

Considering the interactions between objects in the design suggests that what is of interest is failures in *flows* between components. From the review of PSSA techniques in chapter 2 it is clear that this suggests the use of a HAZOP style analysis is the most appropriate. Since we are examining software systems, SHARD would suggest itself as the obvious technique to use. It was seen in chapter 2 how the SHARD analysis classifies information flow failures as service provision (function), service timing, and service value failures. Each of these types of failures must be considered as part of the analysis. The literature survey also discussed how design notations for OO software (such as UML) represent the system using different views, which each view showing different aspects of the design. The view that provides the most useful information for each classification of failures will be different, therefore it was decided to consider the analysis each of the three classes of failure separately.

The discussion of figure 3.3 above, essentially considered only functional aspects of the interactions, that is aspects relating to *what* the interaction accomplishes. It was seen in figure 3.3 how objects can also be viewed as state machines, and how the state behaviour of the object can determine the way objects interact. As part of considering what an interaction accomplishes, the state behaviour of the objects may therefore also be considered. State machines provide a more detailed view of the behaviour of the objects in the system.

Since most systems with safety implications are real-time in nature, it is just as important to consider *when* the interaction is accomplished. In many cases for example, an interaction occurring later than required, could be as bad if the interaction didn't occur at all. These are the temporal aspects of the interactions.

It was noted in chapter 2 that a set of values for data attributes of an object can be represented by a state. Therefore a state transition of an object represents a change in these data attributes. The correct state behaviour of an object can therefore be dependant on the accuracy of the data attributes. An interaction may also involve an exchange of data between objects. These are the value aspects of the interactions, and must also be considered. Although there would seem to be cross-over between function and value aspects of the interactions, a valid distinction

can be made. Value aspects are those where the value of the data itself is the *cause* of the failure. Errors caused by functional changes to the operations which manipulate that data are dealt with as functional aspects.

To attempt to perform SHARD-style analysis on all the possible interactions in any even moderately sized system design would quickly become intractable. In any case, the failures in many of the information flows in the design may have no impact on the hazardous failure modes that the safety analysis is concerned with. Therefore it is crucial to identify firstly which parts of the design are important with respect to the hazardous failure modes. Fault trees were seen in chapter 2 to provide a way of identifying how combinations of failures can lead to some event which is of interest. In this way fault trees can be used to identify the failures in which part of the design could lead to a hazardous failure mode. Those parts of the design identified in this fault tree analysis would then form the subject of further investigation through the use of SHARD. This approach should ensure that all the relevant failures are considered, but also that effort is not wasted on unnecessary analysis.

The remainder of this chapter proposes an approach for performing the analysis of the interactions. Firstly, a simple example system is introduced.

3.4 An Aircraft Stores Management System

The analysis approach introduced in this chapter is best described with the aid of an example software system. In order to illustrate this in a clear and concise way a simplified example will be used. A larger scale, and more realistic example is presented as a case study in chapter 6. The simple example to be used is a hypothetical stores management system (SMS) for an aircraft.

The stores management system contains software which is responsible for the management of stores associated with the aircraft. In the context of an aircraft, a store is essentially anything which is attached to the wing or underside of the aircraft and can be removed. The most common examples of stores are weapons or fuel tanks, but can include other things such as navigation equipment. The stores are attached to the aircraft via *stations*. The purpose of the SMS is to maintain an up to date inventory of the stores on the aircraft. This will include information on the location of each store (which station it is attached to), what type of store it is, and the store's current status. The SMS will also manage the stores by selecting the correct stores to be, for example, jettisoned or released at a particular time. This is done by responding

to commands from the aircraft pilot or co-pilot. In reality, an SMS is a more complex system than described here and will provide more functionality. For the purpose of the use of the example in this thesis however, this description will suffice.

Figure 3.4 shows a UML class diagram for the system described above.

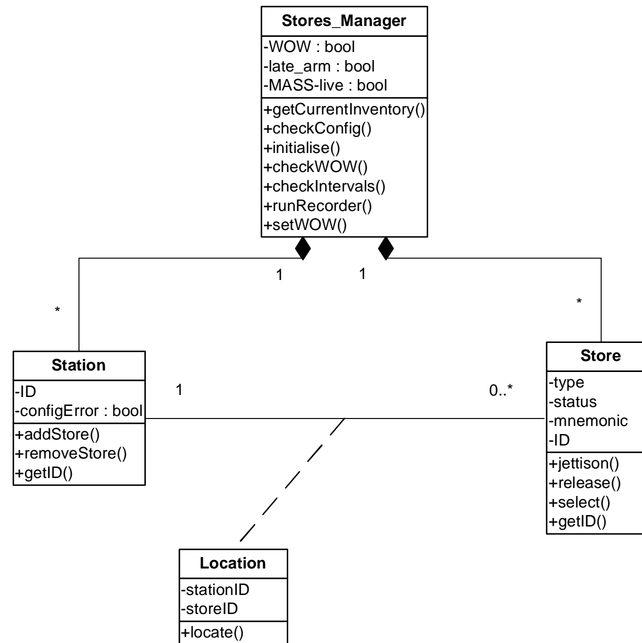


Figure 3.4: UML class diagram for aircraft SMS

Figure 3.5 shows a UML sequence diagram of a normal scenario for the releasing stores function. This scenario involves the release of two stores in sequence, sequence diagrams for alternative scenarios could also be developed.

UML has been chosen as the design notation to be used in the example. This is because, as discussed in 2, UML is by far the most popular notation for designing OO systems. The analysis process is not dependant on the notation used however. As long as the information required for the analysis is available, notation is unimportant.

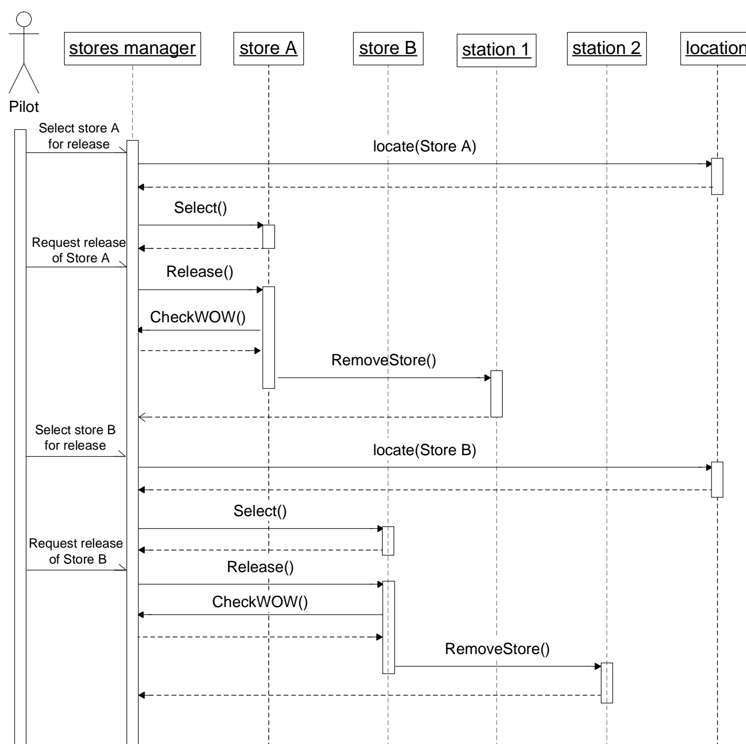


Figure 3.5: UML sequence diagram for release of store

3.5 Analysing Functional Aspects

In this section a method for analysing the safety of the functional aspects of the behaviour of an OO system is described. The purpose of this analysis is to generate a set of hazardous behaviours associated with a particular hazard in the system which requires mitigating. These hazardous behaviours will later be used when deriving a set of safety requirements to mitigate the hazard. The overall process for the analysis of the functional aspects is illustrated in figure 3.6.

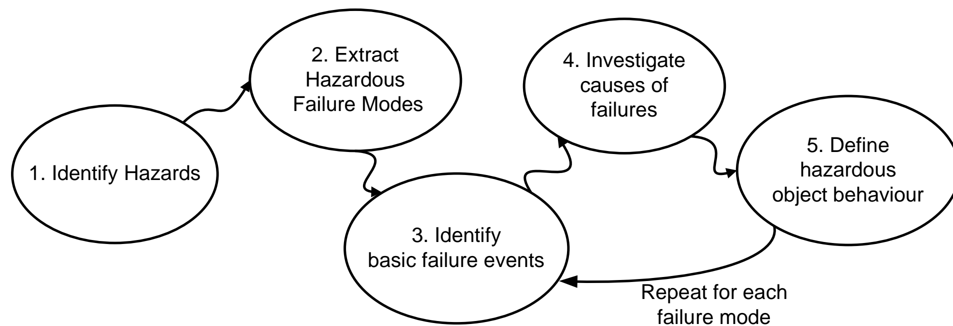


Figure 3.6: Overview of safety analysis for functional behaviours of objects

The five steps in this process are now described using the SMS system from section 3.4 as an example.

3.5.1 Step One: Identify Hazards

The first step involves identifying the hazards that are associated with the system under consideration. These hazards must be acceptably mitigated for the system to be safe. The identification of hazards for a system developed using an OO approach is the same as would be adopted for any safety critical system. As the hazard is a property of the system as a whole, the design paradigm chosen is unimportant. Therefore standard PHI techniques, such as those discussed in 2, can be used to identify system hazards. In practice many of the known hazards associated with a particular system will be recorded in a system hazard log. There will be different levels of risk associated with each hazard, the risk must be reduced to an acceptable level. It is therefore not always necessary to analyse every identified hazard as the risk associated with some hazards may already be acceptably low.

Initial hazard identification for the SMS identifies a number of hazards associated with the system. These hazards include:

- Inadvertent release of store
- Release of store whilst on the ground
- Inadequate temporal separation of store releases
- Unbalanced stores configuration
- Release of incorrect store

The system would need to be analysed for each one of the identified hazards. In order to illustrate the process however, just one hazard will be used. The 'Release of store whilst on the ground' hazard has been chosen for this purpose. This hazard has a potentially high severity associated with it.

3.5.2 Step Two: Define hazardous software failure modes

Having identified the hazards associated with the system in step one, step two involves defining hazardous failure modes relating to the hazards. A hazardous software failure mode has been defined by [84] as "a potential failure within the software which may lead to a system level hazard". At this stage the hazardous failure modes are considered at the level of the software component. Hazardous failure modes for a software component will often be identified based on the results of an FFA used for the system level hazard analysis, as discussed in chapter 2. For the SMS, the hazardous failure modes (HFM) associated with the hazard 'Release of store whilst on the ground' are identified below. As with step one, an existing technique can be used for this step of the analysis, despite the SMS software being OO in nature. As was noted in section 3.2, it is only once the PSSA part of the analysis begins (from step three) that OO requires a different approach.

Hazard - Release of store whilst on the ground

HFM 1 - System fails to prevent a release commanded whilst on the ground

HFM 2 - Release occurs without being commanded by system

HFM 3 - System fails to detect aircraft on ground

Of these identified failure modes, HFM 2 and 3 relate to actuator and sensor failures, respectively. It is therefore HFM 1 which relates to the software system and can be defined as a hazardous software failure mode (HSFM). The HSFM 'Software system fails to prevent release on the ground' will be analysed further in the rest of the process.

3.5.3 Step Three: Identify basic failure events

Step three of the process involves identifying the ways in which the software might fail, and lead to the HSFMs under consideration. As was discussed in section 3.2, it is through sequences of interactions between objects that functionality is achieved by the OO system. This step of the process is therefore concerned with identifying those failures in the interactions which may lead to the HSFMs identified in the previous step. In order to do this it is necessary to understand the interactions that occur. This requires a dynamic view of the system design. A UML sequence diagram can provide the necessary information about what interactions occur between different objects in the system to achieve the desired functionality.

The sequence diagram can be used to work back through the sequence of interactions that occur, and identify which interactions are required to fail in order to bring about the HSFM. A simple fault tree can be constructed to capture the combination of failures which contribute to the HSFM. As was discussed in Chapter 2, there have been a number of attempts at applying a fault tree analysis method to software. The most notable of these is probably Leveson's SFTA technique [38], [39]. In this method Leveson attempted to use fault trees as a way of identifying the contributions of code-level behaviour to software-level hazards. The purpose of this step is different in that fault trees are used in order to identify the failures at the software sub-system level which may contribute to a system hazard. As such the code itself is not considered in the fault tree. additionally, unlike Leveson, the use of fault tree templates is not advocated for this purpose. The fault tree is used to identify the basic events, which are those which relate to failures of individual elements of the design (such as objects or interactions). These failure events require further analysis. This process can again be illustrated using the SMS example.

For the SMS, the top level failure in the fault tree is taken as 'Software system fails to prevent release on the ground' as shown in figure 3.7. In order to construct the fault tree below this failure, it is necessary to consider the UML sequence diagram for release of store as shown in figure 3.5. For this part of the analysis it is only necessary to consider release of a single store, as the general case is being considered. In order to work back through the sequence, the starting point will be the call of the RemoveStore() operation on the station object. This represents the end of the store release sequence. For the top level failure to occur the aircraft must be on the ground, and the remove store operation must be sent. If these events do not occur then neither will the top level failure. The aircraft being on the ground is however considered to be a normal event as it is not a failure (the aircraft is meant to be on the ground at the time). It is therefore represented in figure 3.7 by the normal event symbol. The remove store operation

call being sent *is* a failure event however, as this should not be allowed to happen when the aircraft is on the ground. This event is therefore developed further.

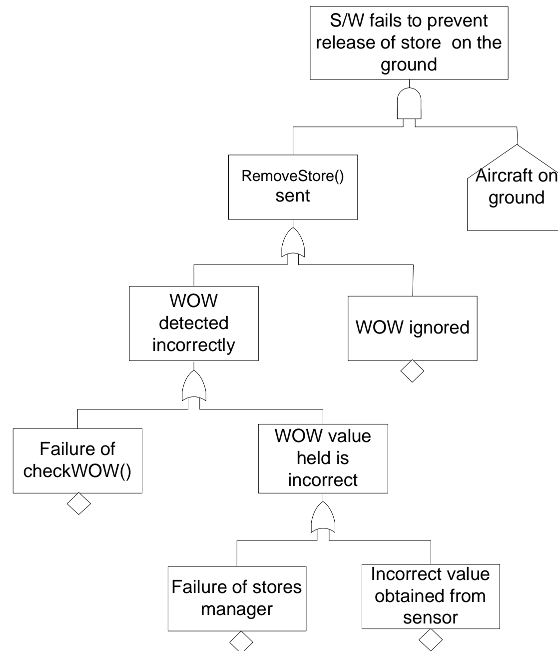


Figure 3.7: Fault tree for SMS HSFM

It can be seen from the sequence diagram (figure 3.5), that before the remove store operation is called, an interaction occurs to check whether there is weight on the wheels (WOW) or not. WOW is a boolean value used to determine if the aircraft is on the ground (true), or not. If this interaction returns true then the remove store operation should not be called. If the remove store operation is sent with the aircraft on the ground then this will either be that the value of WOW is not detected correctly, or that the WOW is detected correctly but is ignored and the store is release anyway. A failure to detect the WOW correctly could either be due to some sort of failure in the interaction that checks the WOW, or that the WOW value was incorrect in the first place. This could either be due to a failure of the stores manager, for which WOW is an attribute, or a failure of the WOW sensor to provide the correct information. The fault tree (figure 3.7), shows how these events relate to the top level failure. It can be seen that four basic events are identified (indicated by a diamond symbol under the event). It is necessary to derive requirements that can mitigate these failures. In order to do this the possible causes of these failures must be investigated. This is done in step four. The fault tree generated in this simple example is quite small. For more complex designs there may be many more levels of decomposition required to identify all the basic events. A more complex design is analysed as part of the case study in section 6.

3.5.4 Step Four: Investigate causes of failures

Having identified the failures that could lead to the HSFM in step 3, it is necessary to derive safety requirements that may mitigate those failures. This requires some further analysis of the design to investigate possible causes of the failures. Step 4 deals with this analysis. It is possible to split this step into two distinct parts, a SHARD-style analysis, which is mainly used for investigating interaction failures, and an analysis of state charts. As a result of this analysis it is possible to identify the hazardous behaviour associated with individual objects in the design. This information can be used to define the necessary derived safety requirements. Firstly the SHARD-style analysis is discussed.

3.5.4.1 Step 4a: SHARD-style analysis

In chapter 2, the SHARD analysis technique was introduced as a software analysis technique based on HAZOP. SHARD is used to assess the suitability of proposed software designs, and to derive safety-related requirements. It is structured around information flows between the components of the system. As such, it is extremely well suited to analysing the interactions between objects in the system. SHARD involves applying a set of guide words to the flows to consider deviation from expected behaviour. The guide words can be interpreted in different ways for different flows. With an OO design there are basically two types of interaction that may occur, an operation call or a signal. For a signal, which is a simpler form of asynchronous inter-object communication, suitable interpretations of the guide words already exist which can be applied (for example MASCOT 3 device output interpretation in [64]). These guide words can be used to determine the potential causes and effects of the deviation, to identify plausible deviations which may lead to one of the failures identified at step 3.

For operation calls, which are the more common form of interaction between objects, there is no existing set of guide word interpretations that is suitable. Therefore a set of suggested interpretations is developed here. The syntax used for an operation is:

`name(parameter-list):return-type`$_{opt}$

These represent the three key elements of the operation call. The name is the name of the operation. The parameter list is a comma-separated list of parameter declarations comprising a name and type. The parameter list may be empty. The return type is optional, and consists of a string containing a comma-separated list of names of classifiers. All three elements need considering in interpreting the SHARD guide words. As part of this thesis, guideword

interpretations for operation calls have been developed, and are shown in figure 3.8. The *guide word* is the SHARD guide word being used. It can be noticed that the service timing group of SHARD guide words has not been used. This is because the timing analysis is carried out separately from the functional analysis, and is discussed in section 3.6. *Deviation* describes the deviation from intent that the guide word suggests. *Responsibility* identifies which of the objects involved in the interaction is responsible for the deviant behaviour. This is recorded as client or supplier. The client is the object which makes the operation call. The supplier is the object that performs the operation. The responsibility is useful in defining the hazardous object behaviour later in the process.

Guideword	Deviation	Responsibility
OMISSION	Operation call not sent	Client
	Return value not returned	Supplier
	Parameter missing	Client
COMISSION	Operation call made when not required	Client
	Unexpected value returned	Supplier
	Unexpected parameter	Client
VALUE	Parameter incorrect	Client
	Return value incorrect	Supplier

Figure 3.8: SHARD guide word interpretation used for operation calls

It is proposed in this thesis that the deviations described in figure 3.8 are applied to those interactions whose failure has been identified as potentially contributing to the HSFM. For these interactions the possible causes and effects of each deviation are identified as well as whether this effect could contribute to the HSFM.

For the SMS, the following failure events where identified at step 3 as potentially contributing to the HSFM:

1. WOW ignored
2. Failure of stores manager
3. Incorrect value obtained from sensor
4. Failure of checkWOW()

Failure number 4 is the failure for which SHARD analysis is required as this relates to the failure of an interaction.[1] The checkWOW() interaction is an operation call between the store

[1] Failure 3 also relates to an interaction, that being between the sensor and the software. As this is a hardware

object and the stores manager object. Therefore the guide words from figure 3.8 are applied. The results of this analysis are shown in figure 3.9.

Interaction 1 –
CheckWOW() : Boolean – *Operation Call*
Client – *Store*
Supplier – *Stores manager*

	Deviation	Cause	Effect	Contribute to HSFM?
OMISSION	1.1 CheckWOW () call not made	Failure of client to send call	WOW value not obtained	Yes
	1.2 WOW value not returned	Failure of supplier to return a value	WOW value not obtained	Yes
COMMISION	1.3 CheckWOW () call made when not required	N/A	N/A	No
VALUE	1.4 Returned WOW value is incorrect	Failure of supplier*	WOW value incorrect	Yes
		Value obtained by supplier incorrect*	WOW value incorrect	Yes

* Failures already identified in FT

Figure 3.9: SHARD analysis of checkWOW() interaction

The causes identified in figure 3.9 will be used in defining hazardous object behaviour in the next process step.

3.5.4.2 Step 4b: Statechart analysis

As was discussed earlier, a statechart can be used to represent how an object evolves over time in response to events. Analysing statecharts is therefore a good way to understand the potential causes of failures. In chapter 2, the work of Gorski and Nowicki on analysis of state charts was discussed. This work can be used to investigate the potential causes of the failures of interest. This can be done in two stages. Statecharts are required for the relevant objects in the system design. The first stage assumes that the object being analysed will behave exactly as specified in the design. This means that there are no mistakes in implementation which cause the object to exhibit faulty behaviour. The aim of this stage is to check that the proposed statechart design will not lead to any of the identified failure conditions. In this way it is possible to check that the statechart design is safe, that is to say that the design if implemented correctly would not contribute to any HSFMs. If the design is found to exhibit any of the failure events then the statechart design must be changed, and the redesign analysed, such that the failures will no

interaction, and is not considered as part of the software design, for clarity, it is not analysed here. Instead this failure will be treated as a simple hardware failure. It should be noted however that a SHARD style analysis of this interaction is possible.

longer occur. For simple statechart designs a manual inspection of design will often be enough. However for more complex designs with many states, manual inspection may not be feasible. In such case a statechart reachability analysis tool [20] can be used.

The second stage of the analysis involves investigating how the object might behave in an unexpected manner, that is to behave in a way other than that specified in the design due to mistakes in the implementation. By doing this, the type of faults that could lead to the failure events can be identified and then constraints defined to prevent them. To investigate this faulty behaviour, the transitions in the state chart can be mutated. The approach adopted by Gorski [17] is to add extra transitions into the state chart to represent possible faulty transitions. Gorski identified five distinct faulty transitions. Given the general form of a transition as e[c]/a, where e is the event that triggers the state transition, c is boolean condition expression which must evaluate to true for the transition to occur, and a is an action that is triggered when the transition fires, the faulty transitions can be defined as:

1. e[c] self-transition
2. not e[c]/a parallel transition
3. e[not c]/a parallel transition
4. e[c] parallel transition
5. e[c]/b parallel transition, where b is an action other than a

These five faulty transitions are derived by considering provision and value failures on each of the transition elements (event, condition and action). As such these five faulty transitions represent a complete set of fundamental faulty transitions that may occur. This approach relies for completeness upon all the possible states and transitions for the object being represented in the analysed state chart. If this were not so, then it would be possible, for example, for an error in implementation to result in the object moving to a state which has not been considered in the analysis. It is also the case that if sequences of transitions are considered, additional faulty transitions are possible through different combinations of the fundamental faulty transitions. Due to the large number of possible combinations, considering sequences of transitions is only really feasible when using tool support. This is considered in section 3.5.4.3. These five mutant transitions are applied to each state relevant to the identified failures, and each outcoming transition of that state in the statechart. As for the non-mutated statechart in the first stage, the mutated statechart is now checked to see if it may lead to any of the failure events. This thesis

proposes that if it is identified that any mutant states can lead to the failure condition then it will be necessary to define constraints to prevent their occurrence in the implemented software. The results of the statechart analysis can also be considered as definitions of potentially hazardous errors in the implementation. It is therefore possible to use this information to specify test cases to specifically look for these hazardous errors in the code. This is particularly powerful when using tool support to help with conducting the analysis. This is discussed in more detail shortly.

For the SMS, the statechart for the store object is considered. The statechart is shown in figure 3.10. The failure events that are being investigated are failures 1 and 2. By examining the statechart the failures can be defined as, 'Object moves to release state when WOW is true.' It can be quite easily seen in this very simple statechart, that the statechart design does not allow this. Once the store is in the *selected* state, a release event causes a transition first to a *WOW checked* state. This ensures the value of WOW is the current one. The transition to the *released* state only then occurs if the WOW is false.

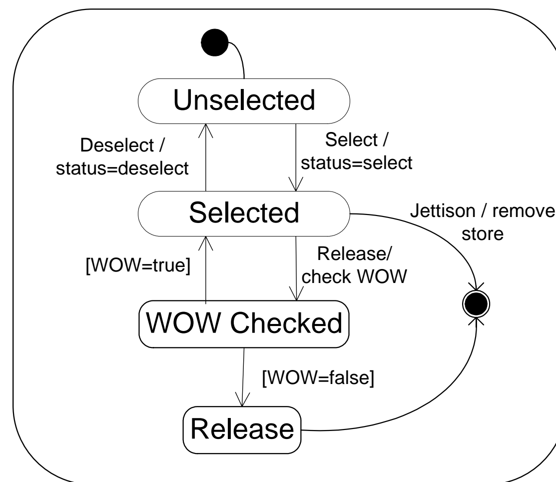


Figure 3.10: UML statechart for the SMS store object

The mutated state chart, with the faulty transitions added, is shown in figure 3.11. It should be noted that the unselected state, and the transitions between unselected and selected have not been included in the analysis as they are not relevant to the failure of interest. This state could be included for completeness, however it should be noted that for more complex state charts, with a large number of states, it is advantageous to limit the number of states considered to those that are relevant in order to keep the analysis manageable. The faulty transitions have been labelled 1 to 5 as appropriate, which corresponds to the five possible mutations defined earlier. Not all the mutations are relevant for each transition, as none of the transitions contains

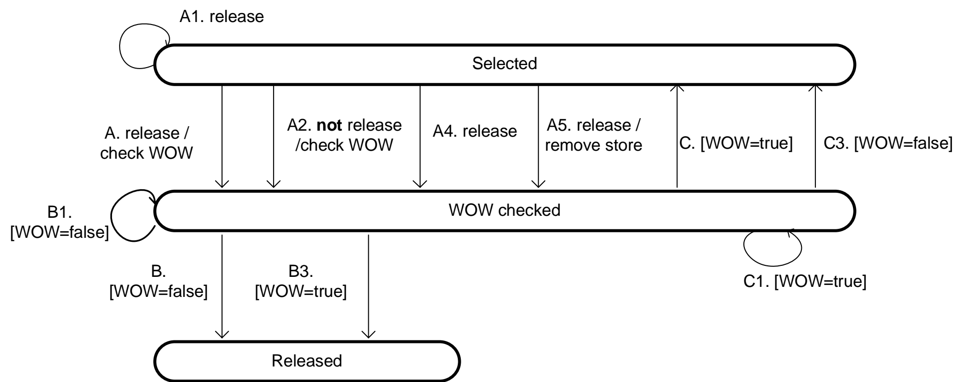


Figure 3.11: Mutated statechart for SMS store object

all of the three basic elements. Each of the faulty transitions is now considered in turn to identify if they could lead to the failure conditions. The results are presented below:

A1 - release *Not Hazardous* Although commanded to release, release state will not be reached.

A2 - not release/checkWOW *Not Hazardous* WOW value is checked prior to release. [2]

A4 - release ***Hazardous*** Release state may be entered without WOW value being known.

A5 - release/remove store ***Hazardous*** Store is removed without required checks.

B1 - [WOW=false] *Not Hazardous* Release state will not be reached, even when WOW is false.

B3 - [WOW=true] ***Hazardous*** release state is entered when WOW is true.

C1 - [WOW=true] *Not Hazardous* Release state will not be reached.

C3 - [WOW=false] *Not Hazardous* Release state will not be reached, even when WOW is false.

It should be noted that all the transitions discussed above represent *incorrect* behaviour, that is they all represent behaviour which deviates from the intent of the design. Not all of the incorrect behaviour is potentially hazardous however (as identified above). Behaviour which is incorrect, but will not lead to a hazard is not of concern from a safety point of view. So the analysis above has identified from many possible errors that could be made, which could be hazardous. This information, along with the results of the analysis in step 4a, will be used in step 5, when defining hazardous object behaviour.

[2]However, although not commanded to release, release state can be reached, so this would probably be contributory to the inadvertent release hazard. This would therefore be picked up when analysing for that hazard.

For more complex statechart designs than that shown in figure 3.10, a manual approach to analysis, as described above, may not be feasible. Even with just a few states and transitions, as in this example, the number of mutated transitions can still become potentially large. Therefore, for this statechart analysis to be feasible, it is desirable that tool support is available.

3.5.4.3 Statechart analysis tool support

Of all the parts of the analysis proposed in this thesis, this state chart analysis has the greatest potential to become intractable for larger systems. To ensure that all parts of the analysis are scalable to larger and more complex systems it was therefore identified that the assistance of an automated tool is required. A toolset developed at the University of York was identified as being able to be applied to meet the requirement for an automated tool. The tools in the toolset have been used to provide assistance in a number of ways. Firstly, they can identify sequences of transitions that may bring about a defined hazardous state, and generate mutations of transitions which may arise due to mistakes in the implementation. These two operations may be performed in combination to identify which mistakes in implementation could lead to a defined hazard, as for the second stage of analysis step 4b. The toolset is described in more detail in [20].

There are two additional functions which are possible with tool support which could not be performed manually. Firstly, the method described in section 3.5.4.2 considers only failures in single transitions. Through the use of the toolset it is possible to consider the consequences of sequences of faulty transitions. Also, an extra function which is provided by the toolset is the ability to perform test data generation. This is done by identifying the input values that would trigger a hazardous transition, and the output values that would result from that transition and inputs. Input and output values are then determined simultaneously. The solutions are written out in a form suitable as input to a test harness tool. This makes it possible to create test cases to check for the existence in the implemented design of any of the hazardous potential mistakes in the implementation.

3.5.5 Step Five: Define hazardous object behaviour

The final step in analysing functional aspects of the system is to define the hazardous behaviour of the objects in the system which must be eliminated to prevent the occurrence of each identified HSFM. This is done based on the information obtained from the previous steps in the process. There are two main sources of this information; the output of the SHARD analysis for step 4a,

and the results of the statechart analysis performed in step 4b.

Each of the deviations, for each of the interactions analysed in step 4a, which were identified as contributory to a HSFM is taken. For each of these, the cause of the failure is used to define the hazardous behaviour of the relevant object. For each HSFM there will normally be behaviour of more than one object which has been identified as hazardous. Each of the faulty transitions from step 4b which were identified as hazardous is also taken, and again can be used to define hazardous behaviour for the relevant object. Once the complete set of hazardous behaviours relating to each HSFM has been identified, it is necessary to consolidate these behaviours. By doing this it is intended to remove duplication, and also to ensure that there are no conflicting hazardous behaviours for any HSFM.[3] The outcome of this will be a set of hazardous object behaviour which can be used later in defining constraints. This will be discussed further in chapter 4.

For the SMS the hazardous object behaviour relating to HSFM 'Software system fails to prevent release on the ground' has been identified as follows:

From step 4a

Deviation 1.1 Store object does not send checkWOW() call prior to removeStore() call

Deviation 1.2 Stores manager object returns no value in response to checkWOW()

Deviation 1.4 Stores manager object returns incorrect value in response to checkWOW()

Deviation 1.5 Hardware failure of aircraft WOW sensor

From step 4b

A4 Store object does not check WOW value prior to release

A5 Store object does not check WOW value prior to release

B3 Store object release when value of WOW is true

These hazardous behaviours can be consolidated into the following 4 for the analysed HSFM:

1. Store object does not send checkWOW() call prior to removeStore() call
2. Stores manager object returns no value in response to checkWOW()

[3] It is not expected that conflicts should be present for any of the HSFMs. A conflict may be for example that it was identified to be hazardous for an object both to do, and not to do, the same thing. If this did occur then it is most likely that there has been an error made in the analysis, or there is an inherent weakness in the design which should be rectified.

3. Stores manger object returns incorrect value in response to checkWOW()

4. Store object releases when returned value of WOW is true

These behaviours will be used in chapter 4 to specify constraints used to prevent this HSFM.

3.6 Analysing Temporal Aspects

In this section a method for analysing the safety of the temporal aspects of the behaviour of an OO system is described. The purpose of this analysis is to generate a set of timing requirements associated with a particular hazard in the system which requires mitigating. The overall process for this analysis is illustrated in figure 3.12.

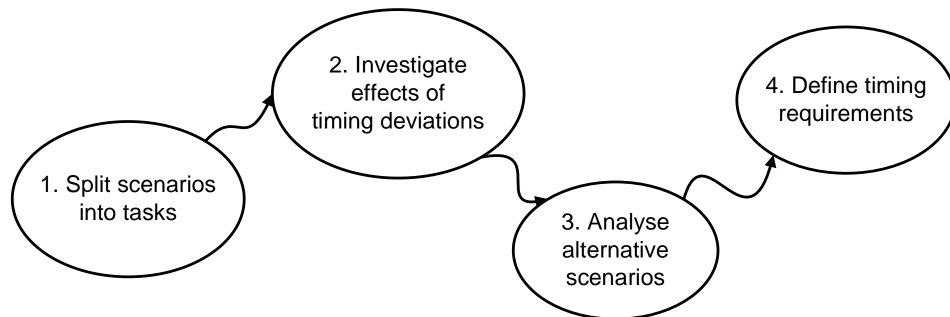


Figure 3.12: Overview of safety analysis for temporal behaviour of objects

There are four steps to this process which are described below. Again, the SMS described in section 3.4 is used as an example.

3.6.1 Step One: Split scenario into tasks

The analysis of temporal aspects is concerned with identifying execution times, separations and priorities for tasks performed by the system. These three properties could affect the safety of the system by causing behaviour which may contribute to a system hazard. A task is an encapsulated sequence of operations that executes independently of other tasks [14]. Therefore, informally, a task can be considered to be made up of a number of interactions between objects in the system which accomplishes some *task*. Although eventually timing requirements will be identified for each relevant interaction, such that safe behaviour can be guaranteed, this thesis proposes that the analysis initially is performed at the level of tasks. The reason for this is that, as discussed earlier, an individual interaction will normally only contribute to the functionality of the system as part of collaborating sequence of interactions. Therefore to understand the

behaviour in order to be able to perform the analysis, it is necessary to consider tasks rather than interactions. A scenario may sometimes contain only one task, however it is often the case that scenarios contain many tasks, which together provide a particular functionality. The identification of tasks in a scenario is best illustrated with an example as for the SMS below.

The scenario that shall be considered here is the same scenario as was under consideration in the functional analysis. The UML sequence diagram for this scenario is in figure 3.5. Initially it is necessary only to consider the release of a single store, so the release of store A only will be considered. The scenario is releasing a store, however it can be seen that a number of separate tasks are required to realise this scenario. These are that the relevant store is located, selection of the store and releasing the store. In fact it is possible to consider the location of the store as part of the selection task, as both actions are part of accomplishing the same aim. Therefore just two tasks have been identified. From an analysis point of view, it is generally better to split a scenario into a small number of tasks, however it must be remembered that tasks must each execute independently of one another. An example of something in this scenario which could not be considered a task is checking WOW, as this is part of the execution of the release task.

So the two tasks identified for this scenario are:

1. **Select store**

   Begin - Pilot selecting store to release

   End - Store is selected

2. **Release store**

   Begin - Pilot commands release

   End - Removal of store from station

These tasks will be used in the next analysis step. Tasks can similarly be identified for each scenario under consideration.

3.6.2 Step Two: Investigate effects of timing deviations

For each of the tasks identified in step 1, the effects of a set of deviations are investigated to identify if the deviations may lead to behaviour which could contribute to a system hazard. The deviations considered are taken from the SHARD guidewords [64]. The SHARD guidewords related to timing are *early* and *late*. These guidewords can be interpreted as deviations from

expected behaviour. Defence standard 00-58 [50], also provides some more detailed timing related guideword interpretations. The interpretation given for *early* and *late* is actions or events taking place before or after they were expected. In addition, 00-58 also provides an interpretation of the guidewords *more* and *less* for response times. These are interpreted as the time from input to output being longer or shorter than required. For this step of the analysis it is necessary to reflect these different deviations by using two different interpretations of the early and late guidewords. The guidewords *more* and *less* do not capture the timing deviations of tasks clearly, and therefore this thesis proposes the introduction of two additional guidewords, *quick* and *slow* to ensure these deviations are captured. The interpretation that is proposed in this thesis for these guidewords when applied to tasks in this step of the analysis is given below.

Quick The execution of a task[4] happens faster than expected

Slow The execution of the task happens slower than expected

Early The task begins too soon after the end of the previous task

Late The task begins too long after the end of the previous task

It should be noted that the terms *faster, slower, too soon* and *too long* used in the guideword interpretations are identified with respect to some 'most desirable' time which has not been explicitly defined here. It is not felt that a more concrete definition is in fact necessary, as there is an implicit assumption in the design that the most desirable is represented. It is therefore deviations from the intent of the design that is of interest.

Each of the deviations is applied to each task. For each deviation it is identified if the behaviour could contribute to any hazardous behaviour as identified in step 1 of the functional analysis. For deviations which could lead to hazardous behaviour, timing requirements which can ensure the behaviour is not hazardous will be defined later in the process.

For the tasks identified in the release store scenario for the SMS, the effects of the deviations were identified as shown below in figure 3.13.

The task deviations which could impact safety are therefore:

1. Select store - Slow
2. Release store - Slow
3. Release store - Early

[4]Execution is considered from the beginning to the end point as described in the task description in step 1.

Task	Deviation	Effect
Select Store	Quick	Not Hazardous (positive effect) – It is desirable that the selection of the correct store occur as quickly as possible
	Slow	**Hazardous** – Delays in selecting the appropriate store for release may delay release
	Early	Not Hazardous – There is no requirement to wait before selecting another store
	Late	Not Hazardous – The pilot will trigger this in response to operational requirements
Release Store	Quick	Not Hazardous (positive effect) – It is desirable that the store be released as quickly as possible when requested
	Slow	**Hazardous** – A delay in releasing a store could be hazardous to the aircraft under certain circumstances
	Early	**Hazardous** – A weapon released too soon after a previous weapon could be catastrophic for some weapon types
	Late	Not Hazardous

Figure 3.13: Timing deviations applied to tasks for the SMS

Timing constraints must be derived which define response times and separations required to ensure these deviations do not occur and contribute to a system hazard. This will be done in step 4.

3.6.3 Step Three: Analyse alternative scenarios

Defence standard 00-58 suggests *before* and *after* guidewords for capturing events happening before or after another event which is meant to precede or follow it. Rather than use these deviations in this way, this thesis proposes a slightly different approach. Before and after guidewords for tasks essentially suggest an alternative scenario, that is a scenario different from the normal, or expected behaviour. It should be noted that there are many potential alternative scenarios for any particular function. This step of the analysis considers this by generating alternative scenarios, which can then be analysed.

The analysis performed in step two considered only the normal behaviour scenario for the functionality being considered. The normal scenario simply represents the normal or expected sequence of tasks which occurs for a particular functionality. When considering the safety of the system, it is important to also consider alternative scenarios which may occur as these could be potentially hazardous, or may lead to a requirement for additional timing constraints. Alternative scenarios can be identified by applying deviations to the normal scenario. The deviations proposed are as follows:

- Task omitted

- Extra task added
- Task occurring concurrently with others
- Task occurring out of order (c.f. *before* and *after*)

There is potentially a huge number of alternative scenarios that could be considered, as the various deviations could be applied in combination, and any number of times. Most of these possible scenarios will be of little interest however, either because they are essentially identical to previously identified scenarios, or because they would realistically never occur. There will also be a number of scenarios where it is clear that there would be no additional contribution to hazards, over that which has already been seen in other scenarios. Therefore most possible alternative scenarios can normally be easily discounted leaving a fairly small number of significant scenarios to consider.

It is useful for this part of the analysis to be able to represent the scenarios clearly and concisely, as well as emphasising the sequential and concurrent nature of the tasks in a scenario. It has been found that the UML notation of activity diagrams can be adapted for this purpose. Activity states in an activity diagram are normally used to model a step in the execution of a procedure. For this analysis, it is proposed that each activity state will be used to represent a task in the scenario.

For the SMS, this analysis is fairly trivial as the scenario that is being analysed (release of store) entails only two tasks. For completeness however the scenario is shown in figure 3.14 along with some example alternative scenarios that may be considered through applying the deviations. In the first alternative scenario, extra tasks have been added to the end of the original scenario. This considers the situation where two releases occur one after the other. In this particular case the effects have already been considered via the early and late key words used in step two, so no additional constraints are required.

A more interesting scenario is number two. In this case the effect of another store being selected, before the previous one has released, is considered. The key thing here is that it is important that the actions of the pilot in selecting the next store does not prevent the current release, as this could be hazardous in situations were release is required for the protection of the aircraft. It is important therefore that the priority in execution is given to the release task. Any request from the pilot for release of a different store must only be dealt with once the current release task has been completed. This may of course have a knock-on effect on the execution times of the tasks, as waiting for the previous release to finish will increase the time for the next tasks

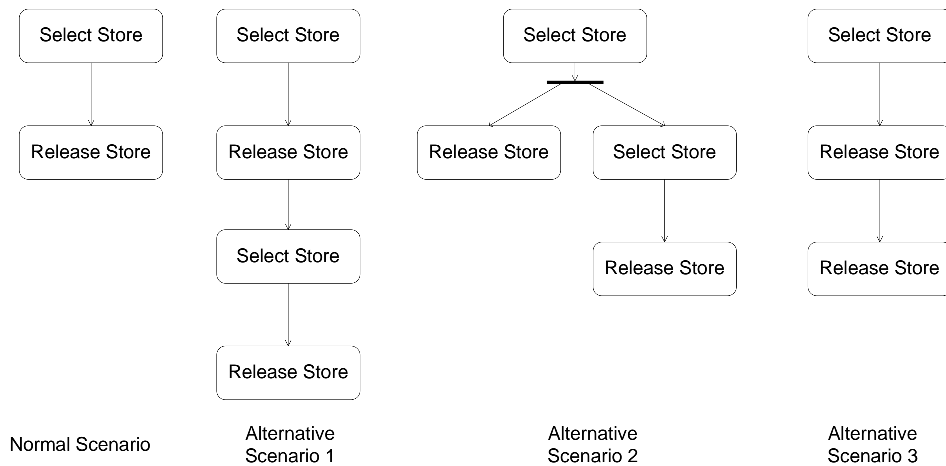


Figure 3.14: Possible alternative scenarios for release of a store

to complete. This will be considered when defining timing constraints.

The final scenario that is considered is where the select store task is omitted. In this case there has obviously been an inadvertent release request either by the pilot, or by the system itself, however this should not result in hazardous behaviour as the release will not occur, as the selected store has already been removed. There are no additional timing constraints generated from this scenario.

3.6.4 Step Four: Define timing requirements

Based on the analysis performed in steps two and three, it is necessary to define timing constraints which state requirements which must be met so as not to contribute to any of the system hazards. This firstly involves defining constraints on tasks whose timeliness was identified as potentially contributing to a hazard. The constraint required will depend on what type of deviation might be hazardous. For those tasks where quick or slow might be hazardous it is necessary to constrain the response time of the task. A minimum response time is required for those tasks where too quick was identified as being hazardous, and a maximum response time, or deadline, is required for those where too slow could be hazardous. For tasks where early or late may be hazardous, minimum and maximum separations respectively between the completion of one task and the triggering of the next, or between an event and the triggering of a task must be specified. Figure 3.15 illustrates how these constraints can be used together to define a safe scenario for the tasks.

In order to specify the requirements, a high level of domain knowledge is required to understand the timing behaviour of the system. It should be noted that in specifying the requirements, it

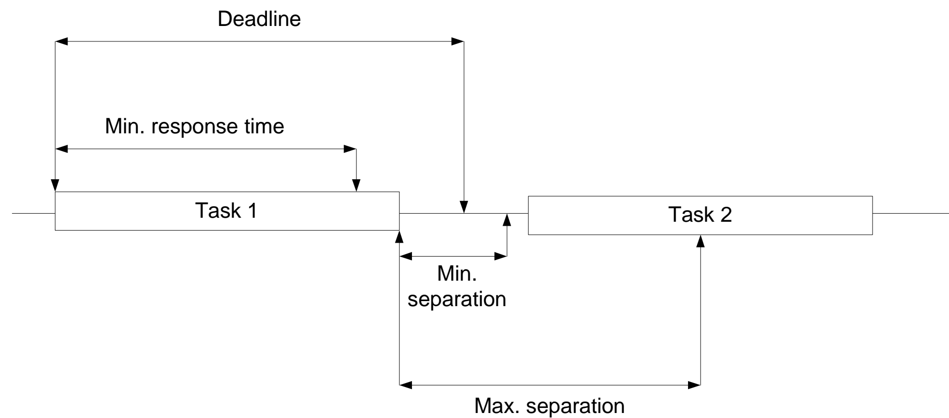


Figure 3.15: Timing constraints on tasks in a scenario

is not intended to produce accurate estimates of execution times for tasks or operations, but to specify the *minimum* requirements for a safe system. Indeed it is desirable that as much flexibility as possible is included when defining the timing constraints, as this makes it simpler to ensure that the requirements can be met.

These timing constraints are defined upon tasks. It was explained earlier that ultimately it is required that timing constraints can be specified on interactions, such that they can be enforced along with the functional requirements which are identified on the interactions. There are some issues associated with this which are discussed below. Figure 3.16 shows a task in the form of a sequence diagram. Sig1 is a signal, which is an asynchronous communication between objects. The signal is the most fundamental interaction between objects and is therefore fairly easy to deal with. Let us consider, however, the interaction Op1. This is an operation call which is an asynchronous communication between objects. When a call is sent from object B to object C, an invocation of an operation of object C occurs. This moves the thread of control temporarily from the calling procedure of object B to the called procedure of object C. Object B regains control when object C returns. The timing for an operation call interaction is the time from sending the call message to the receipt of the returned message. What this would mean is that any timing requirement, such as a deadline, placed on interaction Op1, would implicitly also place requirements on interactions Op2, 3 and 4, as these interactions need to occur before Op1 can complete.

For the SMS, based on the earlier analysis, timing requirements were identified to be required on the following tasks:

Select store Deadline

Release store Deadline

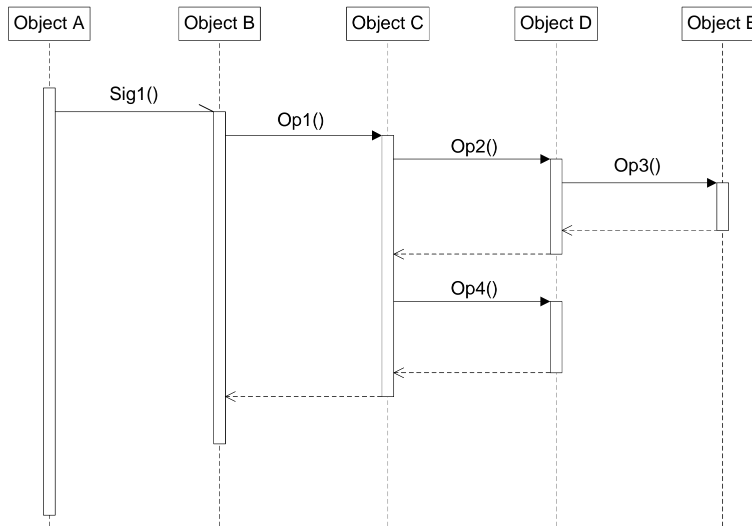


Figure 3.16: Sequence diagram representing a task

Release store Min. separation

Release store Priority

The requirements will be different depending on the type of store being used on the aircraft. However based on an understanding of the performance of typical store types the following requirements can be made:

Select store Deadline 200ms

Release store Deadline 50ms

Min. separation 100ms

Since priority is to be given to the release store task, this may delay a subsequent select store task by as much as the deadline for release of store (in this case 50ms). Therefore the timing budget actually available to be allocated for the execution of operations in the select store task is reduced to 150ms to ensure that the task deadline can still be met. In allocating budgets to specific operations there is normally a trade-off to be made in how much of the budget is assigned to the different operations. This will be based on a judgement of computational requirements of the operations. In this example it is assumed that all operations demand equal timing resource, and the timing budget is shared out accordingly. This leads to the following timing requirements for safety:

locate(store) 70ms deadline

select() 70ms deadline

release() 100ms min. sep, 50ms deadline

checkWOW() 20ms deadline

removeStore() 20ms deadline

3.7 Analysing Value Aspects

The final part of the analysis process involves consideration of the potential contribution to system hazards of errors in the data within the system. The data which will be considered for an OO system can be data attributes associated with an object, it can also be parameters passed as part of an interaction, data returned in response to an operation call, and could also be the value of an object pointer. The aim of this part of the analysis is to define constraints which will control any data whose inaccuracy could contribute to hazardous behaviour. The process used for this analysis contains only three steps as shown in figure 3.17.

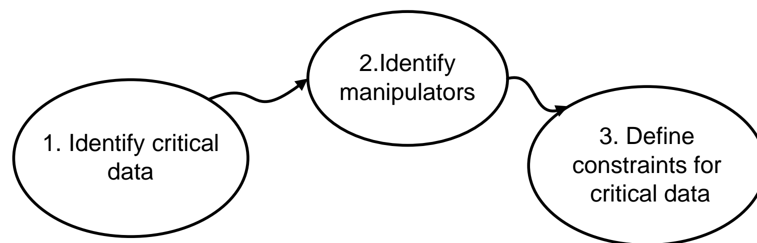


Figure 3.17: Safety analysis process for value aspects

3.7.1 Step One: Identify critical data

The first step of the process involves identifying *critical* data within the system design. Critical data is defined in this thesis as any data whose inaccuracy could contribute to a system hazard. The way in which this can be done is by using a fault tree to investigate causes of hazardous failure modes, and identify failure events relating to data in the fault tree. The advantage of this approach is that such fault trees for hazards of interest in the system will already have been constructed during step 3 of the functional analysis (section 3.5.3). For this first step it is therefore possible to revisit these fault trees and identify the critical data. The type of data, and the object or interaction in the design that it relates to, should also be identified.

For the SMS, a fault tree was created in figure 3.7 for the hazardous failure mode 'Software system fails to prevent release on the ground'. It can be seen from this fault tree that an incorrect value of WOW could contribute to this failure mode. Therefore WOW is identified

as critical data. When fault trees are constructed for hazardous failure modes relating to other system hazards, similar data-related failures can be identified. For example, relating to the 'Release of incorrect store' hazard, selection of incorrect store is identified as a contributing failure, which can be caused by a store being associated with the incorrect station. It can be seen in the class diagram for the SMS (figure 3.4) that store and station are identified by an ID attribute. Therefore, in this case both store ID and station ID would be identified as critical data. There are more items of critical data which can be identified for this system design, however to illustrate the process we will concentrate on these three:

WOW attribute, Stores_Manager

ID attribute, Store

ID attribute, Station

3.7.2 Step Two: Identify manipulators

In order to control this critical data such that it will not contribute to a hazard, it is necessary to understand how this data may be manipulated. With an OO approach, information hiding allows a certain amount of protection for critical data. If an attribute is declared to be private, then it can only be manipulated by operations provided by the object of which the data is an attribute. Reporter operations (also referred to as inspectors) are used to report the value of the data. Transformer operations may alter the value of the data. Transformer and reporter operations associated with the critical data item must be identified. Another way in which data can be used is through being passed as a parameter in an interaction to another object, or returned in a method call. It is therefore necessary to also identify interactions where critical data is passed via interactions. Any of these mechanisms for affecting critical data shall be referred to here as *manipulators*. The information required can be obtained from a UML class diagram or other suitable static view of the system design.

For the critical data identified for the SMS in step one, the manipulators can be identified using the UML class diagram in figure 3.4. Firstly, for the WOW attribute, there is a transformer operation setWOW() which is used to update the WOW value. There is also a reporter operation, checkWOW(). For the store and station IDs, it is a little more complex. The only transformer operation associated with this data is addStore(). This takes a parameter of the store ID and creates an association between that store and the station ID via the creation of a location object. This location object has a reporter operation, locate() which takes a store ID

as a parameter and returns the value of station ID. There are also reporter operations getID() for both station and store. The manipulators identified can be summarised in a table as in figure 3.18 below. These manipulators are used in defining constraints which can be used in ensuring the critical data remains correct.

Manipulator	Type	Parameter	Return
setWOW()	Transformer	WOW	-
checkWOW()	Reporter	-	WOW
addStore()	Transformer	store ID	-
locate()	Reporter	store ID	station ID
getID()	Reporter	-	ID

Figure 3.18: Manipulators identified for critical data in the SMS

3.7.3 Step Three: Define constraints for critical data

Whereas for the function and timing aspects of the interactions it was possible to identify fairly precisely what the necessary constraints would be to control the behaviour, for the data aspects it is much harder. The correct value of the data at any point in time will be dependant on many contextual and environmental factors. For example, when adding a store to a station, it is impossible to specify constraints to ensure that the correct store ID has been associated with the station. This could only be done by checking that the actual store on the aircraft corresponds with the data representation. There are however, certain constraints that can be defined to ensure that the data is valid. That is that the value of the data is a value that would be expected. In this way, if critical data is found to be invalid, then this will be identified as potentially unsafe. Although this can provide some protection, it should be noted that subtle value errors (that is errors which are valid data) will not be detected. These subtle errors can only be mitigated by verifying that the data manipulation performed by the software is correct.

For the SMS, it is possible to define the following constraints on the critical data identified in step two:

WOW Boolean

store ID integer 1$\leq$ ID$\leq$ 1000 **and** exists store.ID=ID

station ID integer 1$\leq$ ID$\leq$ 20 **and** exists station.ID=ID

For boolean data the only check possible is that the value is indeed boolean. For the ID value, more consistency checking is possible. Firstly, it is possible to check that the value is an integer within the expected range. It is also possible to check that it is a valid value by checking that there is indeed a store, or station, which has that value for its ID.

3.8 Conclusions

In this chapter a process has been described for identifying potentially hazardous behaviour in an OO system design. Despite the existence of a number of techniques for analysing OO systems, chapter 2 identified that no systematic and thorough process exists. The process described in this chapter addresses this weakness. The process makes use of existing analysis techniques (such as FTA and SHARD) which have been identified as being the most appropriate for each step of the analysis process. The case study in chapter 6 shows how the process is applied to a large-scale and more complex system design.

The analysis focusses on interactions between objects. This is used to identify failures in the behaviour of the interactions that could contribute to a system hazard, and thus potentially to an accident. In order to prevent the hazards arising it is necessary to define and enforce a set of derived safety requirements (DSRs) on the software design. In the next chapter, the way in which these DSRs are defined and implemented in the form of safety contracts, based on the outcome of the analysis described here, is discussed.

Chapter 4

Safety Contracts

4.1 Introduction

In chapter 3, an analysis process for OO system designs was identified and described. The outcome of this process is a set of behaviours associated with the interactions between objects in the system design, which if present in the implemented software, could contribute to a system hazard. These behaviours can be used to specify derived safety requirements (DSRs) on the software design. The role of DSRs is to define constraints in the system design which must be met by the software in order to guarantee that there will be no contribution to a hazard. By verifying that the DSRs are met, it is possible to conclude that the software is safe in the system context within which the DSRs were derived.

The way in which DSRs are specified for a system can be very important, as the ability to verify the system against those requirements is the key to assuring the safety of the system. In this chapter the use of contracts as a way of specifying safety requirements is proposed. Firstly the reason that a contracts approach to specifying DSRs has been chosen is discussed, before looking at how DSRs, such as those identified from the analysis in chapter 3 may be represented in the form of *safety contracts*. The advantages gained for OO systems from using safety contracts are then examined in more detail. This is done firstly by discussing how the effects of changes to the design may be minimised through using safety contracts. Secondly, how safety contracts make design reuse in safety critical OO systems more manageable is explored.

4.2 The Need for Contracts

In chapter 2 the concept of software contracts was reviewed. A contract is made up of pre and post condition assertions on an object's operation. The precondition must be true when the operation is called, and the postcondition must be true when the operation returns. The contract links the operation supplier with the client objects calling that operation. The supplier guarantees that the postconditions hold if the preconditions have been met. The client therefore can know that the desired outcome will be achieved, without needing to know any information about how this is done. The supplier, in turn, can assume that the precondition will be met and need not be concerned with cases where this is not so.

As well as pre and post conditions, contracts may also contain invariant assertions. Whereas pre and post conditions are assertions on an operation, assertions can also be placed on the object class. Such assertions are known as invariants, or class invariants and must be preserved by all the operations of a class. For each participating object, the contract assertions identify a set of obligations. These obligations specify the behaviour of that object. An object may participate in many interactions, and therefore an object's obligations may arise from many contracts.

The discussion above indicates that the use of contracts could be a useful way to specify DSRs. It was shown in chapter 3 that the key to ensuring safe behaviour in an OO system is to control the interactions that occur between the objects in the system. Contracts can be used to explicitly specify interactions among objects, and therefore can be used in controlling the behaviour of the interactions. Assertions in contracts are normally used to specify the expected behaviour of the participant objects. If contracts are used for specifying DSRs, as this thesis proposes, then their intention is different. The contract no longer specifies expected behaviour, but only that behaviour which is required to ensure the participant objects do not contribute to a HSFM. As such there may be behaviour which is expected of the object which is not specified in the contract, as it is does not impact the safety of the system. There may also be obligations within the contract which are more stringent than would be otherwise required, which appear because they are necessary for safety. The contracts used to specify DSRs will therefore be referred to as *safety contracts* to distinguish them from more conventional software design contracts. It should be noted that both safety *and* non-safety contracts could be used for a system design, as they each serve different purposes. It is of course necessary that the design contracts do not specify behaviour that could breach a safety contract, as this could result in a hazard. This will be discussed in more detail later.

By identifying the obligations on an object arising from the safety contracts in which it participates, it is possible to identify the complete set of safety obligations for that object. These are the obligations which that object must meet to ensure it does not contribute to any unsafe behaviour. Figure 4.1 shows how a set of safety obligations for an object is constructed from a number of contracts. Object A interacts with a number of other objects (objects B to E). In these interactions object A is either the client of an operation (opB() and opD()), or the supplier of the operation (opA1(), opA2(), and opA4()). For the interactions, there may be safety contracts defined, as identified from analysis of the design. For each interaction there may be pre conditions, post conditions, or both defined. For each of the interactions for which object A is a client, object A must meet the preconditions of any safety contract on that interaction. For each interaction of which object A is a supplier, object A must meet the post conditions of any safety contract on that interaction. As object A is involved in a number of interactions, there could be a number of obligations on object A arising from safety contracts. These obligations can be identified for object A as the set of safety obligations associated with object A, as illustrated in figure 4.1. In order to be safe (to not contribute to a system hazard), object A must behave in a manner such that it meets this set of safety obligations. Object A (or the implementor of object A) does not need to have any knowledge of the behaviour of other objects in the system in order to achieve this. The set of obligations can in effect be viewed as a *safe envelope* within which object A can operate and be sure it will not contribute to hazardous behaviour.

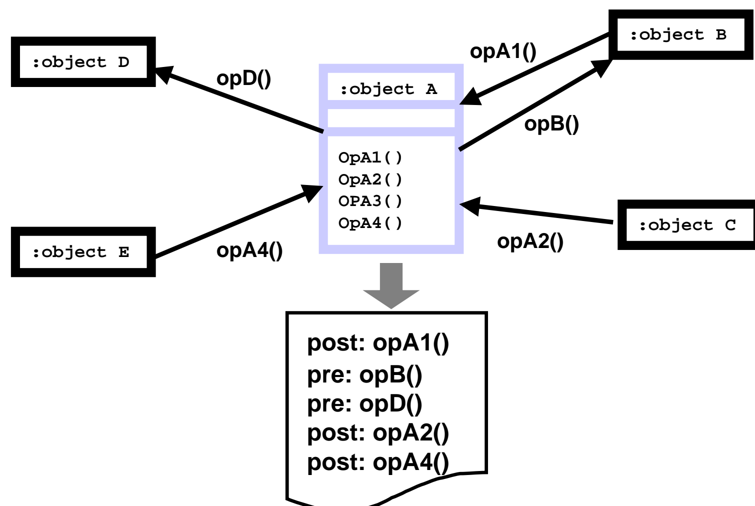


Figure 4.1: Identifying safety obligations for an object

By identifying safety obligations relating specifically to an object in the system, it makes it much easier to cope with changes to the design. It was seen in chapter 2 that robustness to change

is an important feature of OO systems which is often stated as an advantage to adopting an OO approach. Because the safety properties of an object are explicitly identified, those safety properties potentially affected by any change, can be restricted to those which relate to the objects impacted by the change. If the safety properties relating to each object were not so clearly identified, it would be very difficult to identify which properties could be affected by a design change. The use of safety contracts to limit the effect of changes is investigated in more detail in section 4.4.1.

Identifying safety obligations for an object also increases the potential to reuse parts of a design. Again, it was seen in chapter 2 that OO systems are claimed to offer greater reuse potential than other approaches. As the safety obligations required of individual parts of the design are made explicit, it makes it possible to reuse existing design artifacts which provide the required behaviour and meet the safety obligations. Without identifying explicitly the set of safety obligations relating to the objects in a design, it would be very difficult to identify whether an existing design artifact exhibited safe behaviour. The use of safety contracts in reusing elements of an existing design is investigated in more detail in section 4.4.2.

As discussed earlier, the implemented software must behave in such a way as to meet obligations arising from contracts. If this is not the case then the software's behaviour may be unsafe. As noted by Meyer in [46], any violation of a precondition indicates a bug in the client, whereas any postcondition violation indicates a bug in the supplier. Precondition violations, although caused by the client object, could be checked by the supplier. However this approach does not fit in with the philosophy discussed previously. Since the supplier should be able to rely on the client meeting its precondition, it is important that the preconditions are checked by the client. An important question arises as to what happens if an obligation cannot be satisfied during execution. Verification that software meets its obligations can be performed prior to run-time using static analysis techniques or testing.[1] However it is still necessary to consider the effect that failures would have and how they may be dealt with. As was discussed previously, if a precondition obligation is not met by the client of an interaction, then the supplier of that interaction is not obliged to do anything. For safety contracts this situation is not acceptable. It means that if the precondition of a safety contract were not met, a client object may behave in an unspecified manner, which could potentially be hazardous.

For safety critical applications it is necessary that any failures are handled in a safe manner. This means that the system must end up in a safe state even if a safety contract obligation has

[1] Verification methods are beyond the scope of this thesis, but were briefly discussed in chapter 2.

failed to be met for some reason. This thesis contends that this can be dealt with through a safety contract approach, however it requires that an additional clause be introduced into the contracts to specify acceptable behaviour should part of a contract not be met. The idea of an *otherwise* clause was first introduced by Cliff Jones as a way of presenting error-tolerant specifications[2]. Jones suggest multi-level specifications of the form:

attempt S1 **otherwise** S2 **otherwise**...Sn

Where S1, S2 and Sn are component specifications. These specifications contain rely and guarantee conditions (which can be considered for our purposes to be analogous to pre and post conditions respectively[3]). For the specification above, the guarantee of S1 must be achieved if the rely of S1 is satisfied, otherwise the guarantee of S2 must be achieved if the rely of S2 is satisfied, and so on.

In this thesis these ideas are adapted to make it possible to specify *failure-tolerant* safety contracts. In such contracts, the objects attempt to meet pre and post condition obligations of the safety contract. If any obligation is not met, then the otherwise condition must be met. The otherwise condition will specify the obligation required to reach a safe state (or the safest state possible), that is a state where system hazards do not exist. It should be noted that by not meeting safety contract obligations, the system has already failed, therefore the system will not in such circumstances function correctly, the key thing is however that the system has remained safe. The otherwise condition is illustrated in more detail in section 4.3.2, along with an example.

4.3 Defining Safety Contracts

In chapter 3 an analysis process was used to identify potentially unsafe behaviours of the software in a system. The behaviour of the software must be controlled such that the unsafe behaviours do not occur. This can be achieved through defining DSRs which specify constraints on the software's behaviour. This thesis proposes that these DSRs be incorporated into the design as safety contracts such that when the design is implemented it is possible to ensure that the software will not exhibit unsafe behaviour.

The SMS example from chapter 3 will be used to illustrate how DSRs may be derived from

[2]The author feels that *failure-tolerant* would be a more appropriate term, since errors that do not lead to failure do not need to be dealt with in this way

[3]Although the terms are analogous, they are not identical. Unlike pre and post conditions, rely and guarantee conditions can be used to capture assumptions about the state of the entire system.

the output of the hazard analysis process. The following hazardous behaviour was identified in section 3.5.5 as relating to the HSFM 'Software system fails to prevent release on the ground':

1. Store object does not send checkWOW() call prior to removeStore() call
2. Stores manager object returns no value in response to checkWOW()
3. Stores manger object returns incorrect value in response to checkWOW()
4. Store object releases when returned value of WOW is true

The DSRs required to constrain this hazardous behaviour can therefore be defined as:

1. checkWOW() call must be sent by store prior to removeStore() call
2. A value must be returned in response to checkWOW()
3. Return value of checkWOW() must be correct
4. removeStore() call must only occur if WOW is false

In addition to these functional DSRs, there are also DSRs relating to timing. These requirements were identified in section 3.6.4 as:

locate(store) 70ms deadline

select() 70ms deadline

release() 100ms min. sep, 50ms deadline

checkWOW() 20ms deadline

removeStore() 20ms deadline

These requirements must also be captured as safety contracts. The final set of DSRs arises from the value aspects of the analysis process. These were identified in section 3.7.3 as:

WOW Boolean

store ID integer $1 \leq$ ID$\leq$ 1000 **and** exists store.ID=ID

station ID integer $1 \leq$ ID$\leq$ 20 **and** exists station.ID=ID

This thesis proposes to represent these DSRs in the form of safety contracts. This involves representing the DSRs in the form of pre and post conditions and invariants. The safety contracts must also contain an otherwise condition. For the hazard of 'Release of store whilst on the ground' which was considered for the SMS in chapter 3, the safest state in response to system failure would be to not allow a release. This condition must be implemented as part of the safety contract. Although this condition could lead to operational difficulties, it is the only acceptable response if a release on the ground might be possible as a result of a failure. For some hazards and for some systems, the associated fail safe state may not be as easily defined as in this example. The dynamic behaviour of the system, such as could be represented in a UML sequence diagram, can be used to identify where safe states exist for the relevant functionality. In some cases a fail safe state may not even exist. In such circumstances, the safest response must be identified which minimises the risk from the hazard.

Before the safety contracts can be defined, a suitable notation in which to represent them must be identified.

4.3.1 Notation for Safety Contracts

In order to represent safety contracts in a clear and unambiguous way, a notation must be defined which can capture the necessary safety properties for the OO system under consideration. In chapter 2, various approaches to representing safety properties such as [28], and [13] were discussed, however these do not provide support for contract specifications. Chapter 2 showed how contracts can be represented in the Eiffel language [47], and there are other languages in which contracts can be defined. These could be used for defining safety contracts on object classes and operations. The safety contracts that are defined should become part of the design specification, in this way the safety properties are independent of any implementation decisions. Using languages such as Eiffel have the distinct disadvantage of making it difficult to use the safety contract constraints, if implementation is not carried out in that language. It is for this reason that OCL [60] has been identified as a potential way of representing safety contracts. OCL was introduced in chapter 2 as a formal constraint expression language. OCL is a modeling language and *not* a programming language, therefore OCL expressions are guaranteed to be without side effect. All implementation issues are out of the scope of, and cannot be expressed in OCL. This is an important advantage of OCL over other notations such as Eiffel. It was noted in chapter 2 that OCL is often used for describing constraints on UML models. This is because OCL is highly compatible with UML and has been developed via the OMG, who

also have responsibility for UML standardisation. This is an important feature as it makes it easier to specify the DSRs as part of the design itself, rather than being a separate entity. This means that DSRs are communicated more easily to the system developers. It was emphasised in chapter 3 that the safety process should be independent of notation as far as possible. The use of OCL for specifying safety contracts does not affect this, as UML is not a requirement of using OCL.

The current specification for OCL provides the mechanism required to represent most of the safety properties of interest, including invariant and pre and postcondition constraints. Some examples of safety constraints in OCL are given later. There are certain features of safety requirements which require further consideration if they are to be represented successfully in OCL. Firstly, as was seen in chapter 3, safety requirements often refer to messages sent between objects. OCL provides a mechanism for specifying messages. It can be specified that communication has taken place during the execution of an operation using the hasSent operator (ˆ). If the communication is an operation, then there may be a return value which needs to be constrained as part of the safety contract. This can be accessed using the result() operation of the OclMessage type. It is possible to check that a result was returned using the hasRetuened() operation. Thus, in an example taken from [60], OCL could be used to specify the following. Where getMoney() is an operation on an object of type Company that returns a boolean, it can be specified for the operation giveSalary() that a call is made to getMoney(), that the call is returned, and the the result of the call is true. This is done as shown below:

```
context Person::giveSalary(amount:Integer)
post:let message:oclMessage=company^getMoney(amount) in
message.hasReturned()      --getMoney was sent and returned
and
message.result()=true      --the getMoney call returned true
```

This is all standard OCL, as defined in [60]. Chapter 3 identified temporal aspects of the system behaviour as also being an important part of safety requirements. Unfortunately the OCL 2.0 specification does not provide a way of specifying such features. An extension to OCL, OCL/RT, for modeling real-time systems has been proposed in [8], which provides a way to specify deadlines and delays in OCL. This uses a new primitive data type *Time*, which is defined to represent the global system time, to provide a mechanism for specifying deadlines and delays.

Deadlines can be specified using OCL/RT in the following manner:

```
contextTypename::operationName(param1:Type1,...):ReturnType
pre:...
post:Time.now<=Time.now@pre+timeLimit
```

In this expression **now** is an attribute of **Time** which shows the current time, **Time.now@pre** is the value of the current time when the precondition was evaluated, and **timeLimit** is a variable used to define the deadline.

OCL/RT also defines an *event model* with three types of event, CallEvent, StartEvent, and TerminationEvent. A CallEvent is raised when a sender issues a call to an operation, a StartEvent is raised when an object is about to start executing an operation, and a TerminationEvent is raised when the execution of the operation is finished. These events can be used in OCL/RT to specify temporal properties such as delays, as shown below:

```
contextTypename::operationName(param1:Type1,...):ReturnType
pre:  LastCallEvent.at+timeLimit<=Time.now
post:...
```

In this expression LastCallEvent.at is the time at which the last CallEvent was raised, this corresponds to the last time the operation was called. Again timeLimit is a variable, used in this expression to specify a minimum delay between calls to the operation. Events can also be used in a similar way to specify properties such as timeouts.

Using the UML 2.0 OCL Specification, along with the OCL/RT extensions, it is possible to represent safety requirements in the form of contracts.

4.3.2 Safety Contracts for the SMS

The DSRs identified in section 4.3 can now be represented as safety contracts using OCL. The DSRs relate to various operations of different classes of objects within the SMS system. To illustrate the use of OCL for defining safety contracts it is only necessary to develop contracts for one operation. A full development of safety contracts from DSRs is performed as part of a case study in section 6. A safety contract shall be developed for the Release operation of the Store class. The contract on this operation illustrates many of the interesting properties of a safety contract. From the DSRs identified in section 4.3, the following can be seen to relate to the release operation.

- checkWOW() call must be sent by store prior to removeStore() call

- removeStore() call must only occur if WOW is false
- release() 100ms min. sep, 50ms deadline

The first DSR requires that a message is sent, which can be done using the 'has sent' operator as part of the postcondition for release. The second DSR can be specified using the result operation of message. The third DSR can be specified using constructs from OCL/RT. The resulting safety contract for the the release operation can thus be specified:

```
context Store::Release()
pre:  previousRelease.at+100<=Time.now
post:  let message:OclMessage=stores_manager^checkWOW() in
message.hasReturned()
and
message.result()=false
and
Time.now<=Time.now@pre+50
```

Note that the CallEvent previousRelease must be defined within the context of Store.

It was identified previously that the otherwise clause required for this interaction is not to allow a release. This must be represented as part of the safety contract. The otherwise clause must be met both in the situation where the precondition is not met, and also when the postcondition is not met. It is the client object which has responsibility for meeting the precondition. If the object cannot meet all the preconditions for the release operation then it must ensure that a release does not occur (as defined in the otherwise clause). This can be achieved by specifying the client object does not call the release() operation when the preconditions cannot be met. It should be noted here that once the release call is made by the client there is an assumption that the preconditions of the safety contract have been met, as the supplier object would not (and indeed, in general could not) check the precondition.

The supplier object has responsibility for meeting the postcondition. If the supplier object cannot meet all the postconditions then it too must meet the otherwise clause and ensure that a release does not occur. In this case, this can be achieved by ensuring that removeStore() is not called on the relevant station, the status of the store should also be set to fail. This will ensure that even once release() has been called, if, for example, the supplier will not be able

to guarantee a timing requirement in the postcondition can be met, or the checkWOW() call fails, a safe outcome is still guaranteed. It should be noted here that, as in this case, once the otherwise clause has been used, the system has failed to achieve what it was intended to achieve, *however* it will have remained safe, which is the priority for a safety related system.

The otherwise clause, either for the pre or post condition, can be included as part of the safety contract. OCL can be used to represent these otherwise clauses in the same manner as for other constraints. To indicate that they are an otherwise clause to the contract, the constraint **otherwise** shall be used. This is not a standard part of the OCL specification, but is introduced here as a way of representing otherwise clauses in a safety contract. The safety contract for the release operation, with otherwise included, is as shown below:

```
context Store::Release()
pre:  previousRelease.at+100<=Time.now
otherwise:  store^release()=false
post:  let message:OclMessage=stores_manager^checkWOW() in
message.hasReturned()
and
message.result()=false
and
Time.now<=Time.now@pre+50
otherwise:  station^removeStore()=false
and
status=fail
```

The use of the otherwise approach to dealing safely with failures, as described above, relies upon the objects involved in an interaction checking that safety contract obligations have or have not been met. If the conditions of the safety contract are verified prior to runtime, and no verification of these conditions is done at run-time, then the otherwise clause serves no useful purpose. In such situations any failures leading to safety contract violations not identified prior to runtime, have the potential to manifest themselves as hazardous failures.

4.4 Utilisation of Safety Contracts

This section examines the way in which safety contracts, such as those developed in the previous section, may be used to assist in verifying that an OO design is safe. In particular, the way in

which safety contracts support changeability and reuse of OO design artifacts, whilst ensuring the system remains safe, is investigated. Through this, it can be seen how the use of safety contracts allows maximum benefit to be achieved from OO features such as encapsulation and inheritance without prejudicing the safety of the system.

The first step in ensuring the correct behaviour of an object in the system is to identify the set of safety obligations on the object's class arising from the safety contracts which are in place. As was discussed previously, the safety obligations on a class in a system design will arise as a result of safety contract constraints from many interactions involving that class. Once the safety contracts have been defined within the system design, it is therefore necessary to explicitly identify the safety obligations associated with each class, in order to ensure that all necessary obligations are assigned to the correct class.

To identify safety obligations for a class, the first step is to identify all the interactions in which the class is involved, either as a client, or a supplier. This can be done using dynamic design views of the system. For the SMS, many interactions for the store class can be identified from the sequence diagram in figure 3.5. For a complete set of interactions, the other sequence diagrams containing store objects would also need to be considered. From figure 3.5, it can be seen that the store class is a client for the checkWOW() and removeStore() interactions, and is the supplier of select() and release(). The safety contracts for these interactions are used to define the safety obligations for the store class. These obligations will consist of the preconditions of the safety contracts of the client interactions, and the postconditions of the supplier interactions. An example of how these obligations could be captured for the store class in the SMS is shown in figure 4.2. It should be noted that this example is incomplete. Since all the necessary analysis of the SMS has not been presented in this thesis, only the obligations which have already been generated are included. To be complete, the table would need to contain the obligations for all interactions, and the safety contracts would need to developed based on analysis of *all* identified system hazards. A complete example is given in the case study in chapter 6.

The table in figure 4.2, firstly identifies the system for which the obligations are identified. This is very important as the safety obligations captured in the table relate to a specific design model. If the system design is changed, then the safety obligations may not be the same. This is discussed in more detail in section 4.4.1. The class within the system design which is being considered is then stated. The supplier interactions are then identified, along with client interactions. For each of these interactions, any relevant safety obligations, otherwise clauses,

System: SMS Aircraft X v.1.2			
Class: Store			
Supplier Interactions	**Safety Obligations**	**Otherwise**	**Assumptions**
release()	let message: OclMessage=manager^checkWOW() message.hasReturned() **and** message.result=false **and** Time.now<=Time.now@pre+50	station^removeStore()=false **and** status=fail	WOW status does not change between check and release
select()	Postcondition from safety contract for select()…		
Client Interactions			
checkWOW()	Precondition from safety contract for checkWOW()…		
removeStore()	Precondition from safety contract for removeStore()…		

Figure 4.2: Safety obligations identified for the store class

and assumptions are recorded, based on the hazard analysis previously performed.

In figure 4.2, only the release() interaction has been completed, based on the results of the analysis from chapter 3. For this interaction the safety obligations are obtained from the postcondition of the safety contract that was defined in section 4.3.2. The otherwise clause states the required response should those safety obligations not be met, and was again defined in section 4.3.2. The assumptions captures information, relevant to the interaction, that was assumed to be true when the hazard analysis of the system was performed. This could be assumptions made about the system, such as it's operational role, or equipment specifications, or it could be assumptions made about the operating environment of the system. The assumptions are important as the safety obligations can only be used to assure the safety of the software when the assumptions hold. Many assumptions will also be captured in the design documentation for the system under consideration. This emphasises the importance of recording exactly which system design the obligations apply to.

Safety obligations can be explicitly defined in this way for each class of objects in the system design. When an object is implemented in the software, it must behave in such a way that all of the safety obligations are met. It is therefore possible to view the set of safety obligations as a *safe envelope* within which an object of the class may operate. So long as the object's behaviour stays within this safe envelope, the object will not contribute to any of the identified

system hazards. The fact that it is possible to reason about the safety of each object in such a way, brings many advantages when changes are made to the design of the system, and also when reusing existing design elements in another system design. In the rest of this chapter the use of class safety obligations when changing or reusing a design is investigated in more detail. Since maintainability and reuse are stated as key advantages to using an OO approach, it is extremely desirable that these properties can be achieved in an efficient, yet safe, manner for safety related systems.

4.4.1 Supporting Design Change through Safety Contracts

It is to be expected that many changes will occur to the design of a system throughout the development process. It is possible therefore, that changes may occur to a system design at a time after safety requirements have been defined as contracts in the design. As mentioned previously, OO systems offer improved stability in the face of system changes, that is that if there is a change to the system, then the affects on the design should be localised to specific elements of the design, rather than effecting the entire design structure. This can mean that the cost associated with making design changes can be greatly reduced in comparison with some non-OO designs.

As was discussed in the previous chapter, when using OO designs for safety-related applications, it is necessary to analyse those designs to ensure that they will not contribute to system level hazards. Any changes made to the system could affect the potential ways in which the system might contribute to the system level hazards. This means that, as a result of any changes made to the system, the altered design must be re-analysed to ensure that the system remains safe. Analysing the design for a system can be a large and onerous task. If, as a result of changes to the design, re-analysis of the entire system was required to ensure the system was still safe, then the ability to limit the effects of design changes through the use of an OO approach, would not be achieved for a safety related system. It is important, therefore, if the benefit of improved stability to change is to be realised for safety related systems, that the re-analysis that is required as a result of any change, can be localised to those specific design elements affected by the change rather than the whole system. In this section some different change scenarios which could affect an OO system design are considered. For each of these scenarios, it is shown how safety contracts can be used as a mechanism for limiting the effects of that change on the resulting amount of analysis which is required to ensure the system remains safe.

4.4.1.1 Changes to a Class Design

The first change scenario that will be considered is a change made to the design of a class in a system. Changes covered under this scenario include changes to the class state behaviour, changes to existing operations, or introducing additional functionality to a class. These changes may all result in changes to the interactions occurring between objects of the class, and other objects in the design. Before the effects of changes to interactions are investigated however, a simpler case must first be considered.

As described previously, DSRs captured in safety contracts on interactions are used to ensure that an OO system will not contribute to system hazards. If a change made to a design for a class does not change the interactions present in the original design, then the DSRs, derived previously through analysis of the original system design will remain valid. The safety obligations for the class, arising from the safety contracts, will therefore remain the same. It must be shown that the altered class design can still meet the safety obligations placed upon it. The class may well behave differently than it did before, and indeed its behaviour may not be as good in terms of, for example, response times as it was previously. As long as the class still behaves in such a way as to remain within the safety envelope described by its safety obligations, then the class will not contribute to a system hazard.

To determine that an altered class design will still meet its safety obligations will require some re-verification of that class, such as unit testing of the class instances. The amount of re-verification required should be relatively small however, and will be limited to verifying only the instances of the class whose design has changed. This is only possible because of the use of safety contracts. Using contracts ensures that changes within one class in the design, do not have an effect on other classes. This is a simple, if fairly trivial, example of how the impact of change can be limited through a safety contract approach.

More interestingly, changes to a class design will often result in changes to the interactions which occur between that class and others in the system. This could be, for example that new operation calls are introduced into the design. In this case, the safety obligations obtained from the safety contracts derived from the hazard analysis performed on the original design will no longer suffice. Any changed interaction may introduce new ways in which a hazard may occur at the system level. Any changed interactions must therefore be analysed to see if they may contribute to a HSFM, and a safety contract defined for the interaction as necessary.

The analysis performed would take the form of SHARD-style analysis of the interaction (as

defined in section 3.8) to identify if functional failures in the interaction could lead to a HSFM. Any impact on timing behaviour caused by the interaction can be analysed by identifying which tasks the interaction is involved in. The output of the timing deviation analysis (defined in section 3.6.2) will identify if these tasks could impact the safety of the system. If so, timing requirements can be derived for the interaction (as in section 3.6.4) to ensure it does not contribute to the hazardous deviation. Finally, it is necessary to check if the interaction is a manipulator of any critical data in the system design (see section 3.7.2).

This analysis may result in DSRs which must then be reflected in a safety contract. This will result in new safety obligations upon the relevant classes. Although analysis of the changed interaction has been required in order to identify safety requirements for that interaction, it is not necessary to analyse any other elements of the system design. The safety contracts that existed upon other interactions, which have not been affected by the change, remain unaltered. If HSFMs had not been decomposed into safety contracts, the impact of the change on the safety of the system would be unclear, which could potentially require that large parts of the system be reanalysed to check that a HSFM could not be brought about.

An example from the SMS shall now be used to illustrate more clearly how a design change may be handled. It is decided that the design must be changed to ensure that the released store is deleted from the location object for the station to which it was attached. This change has been reflected in the UML sequence diagram for release of store as shown in figure 4.3. It can be seen that a deleteStore() interaction between station and location objects has been added when the store is removed by the station.

SHARD analysis (shown in figure 4.4), identifies a number of ways in which the operation may lead to a HSFM (in all cases an unbalanced stores configuration). DSRs will therefore be required to prevent the HSFM occurring. These will be captured in a safety contract for deleteStore().

The new interaction (deleteStore()) is part of the release store task. Reviewing the results of the timing analysis in section 3.6.4 reveals there to be a deadline, and a minimum separation requirement on this task. The deleteStore() operation is not affected by the minimum separation requirement, however, it does need to be budgeted as part of the deadline of 50ms. This will place a timing requirement on the deleteStore() operation, which will be captured as part of the safety contract. It will also have an effect on the other operations in the task, as their timing requirements must be altered to accommodate the execution of deleteStore(). This will lead to more stringent safety obligations (tighter deadlines) on those operations.

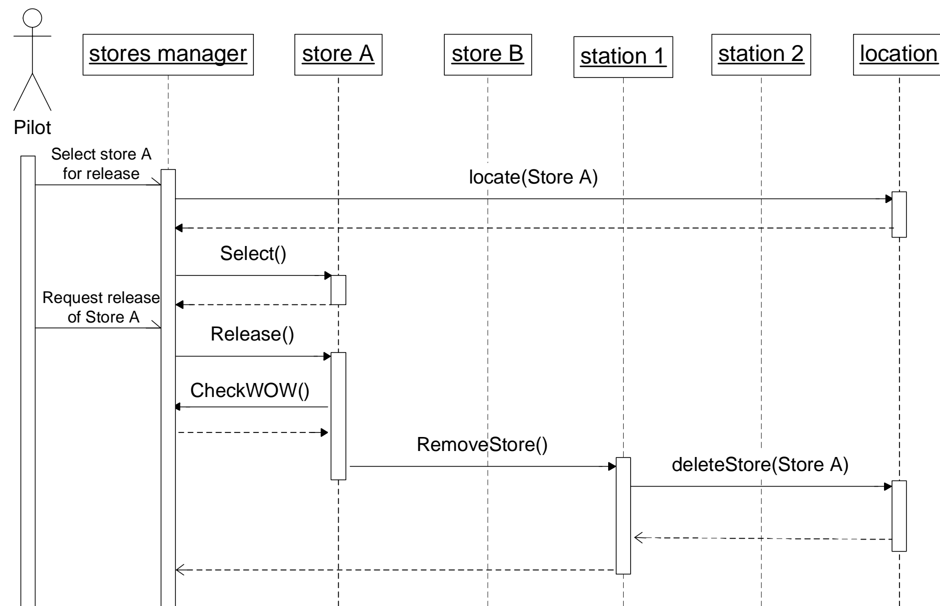


Figure 4.3: Changed UML sequence diagram for release of store

Of the critical data identified in section 3.7.1, the deleteStore() operation can be identified as a manipulator of store ID. deleteStore() is a transformer which passes the store ID as a parameter, a constraint similar to that defined in section 3.7.3 is therefore required as part of the safety contract. No other parts of the analysis performed on the original design need to be revisited, the safety contracts defined as a result of the original hazard analysis remain valid.

4.4.1.2 Introducing New Classes

Another change scenario, is the introduction of a new class into a system design. Once again it will be seen that using safety contracts to specify DSRs means that the effect of the change on the safety of the system is more easily identified, and the re-analysis required can be minimised. When a new class is introduced as part of the design, it is the interactions of that class with other classes in the design which is of interest. It is necessary to identify if the interactions involving the new class could contribute to a HSFM. The techniques for analysing these interactions were discussed previously. For many interactions in which the newly introduced class instances are the client of the interaction, a safety contract may already exist for the operation being called. For these interactions, the preconditions of the contract will become safety obligations upon the new class. In such cases the analysis will be needed to check for additional requirements only. For interactions in which the new class instances are suppliers of the interaction, a full analysis of the interaction is required since the interaction did not form part of the original analysis. Once again the use of safety contracts has ensured that the reanalysis required has

deleteStore(Store) : – *Operation Call*

Client – *Station*

Supplier – *Location*

	Deviation	Cause	Effect	Contribute to HSFM?
OMISSION	1 deleteStore () call not made	Failure of client to send call	Inventory information not updated	Yes – Unbalanced stores config.
	2 Store parameter missing	Failure of client to specify parameter	Inventory information not updated	Yes – Unbalanced stores config.
COMMISION	3 deleteStore () call made when not required	Failure of client	Inventory information incorrect	Yes – Unbalanced stores config.
VALUE	4 Incorrect Store parameter sent	Failure of client	Unknown	No

Figure 4.4: SHARD analysis of deleteStore()

been localised to the new class, with limited effect on the other classes.

When considering introducing new classes to an OO system design, the inheritance mechanism is very important. As was discussed in chapter 2, inheritance is one of the key characteristics of OO systems, allowing new classes to be created from existing ones. The subclasses created inherit all the attributes and operations from the parent class. Additional attributes and operations may be added by subclasses if required. This concept is sometimes referred to as *programming by difference* as it allows new classes to be produced without starting from scratch. This is a very useful feature of OO designs which must be exploitable in a safe manner for safety related systems.

4.4.1.3 Using Inheritance

It is important when using inheritance, that the subclasses created are substitutional for instances of the parent class. This ensures that as far as anything using the objects of the parent class can tell, the objects of the subtype behave the same as those of the parent. A set of subtyping rules which ensure this is the case was developed by Liskov [42], these rules became known as Liskov Substitution Principle (LSP). In [46], Meyer explained how contracts can be used to enforce these principles, by subclasses inheriting a contract from their parent. When creating the contract for the subclass, it is permissable to weaken the preconditions or strengthen the postconditions inherited from the parent class if this is necessary, however the opposite of this is not permitted. This ensures that LSP is followed. In this thesis it is proposed that safety

contracts may be inherited in a similar manner to that suggested by Meyer for design contracts. These ideas are best illustrated with an example. Once again the SMS will be used for this purpose. It is only necessary here to consider one small part of the design, as shown in figure 4.5.

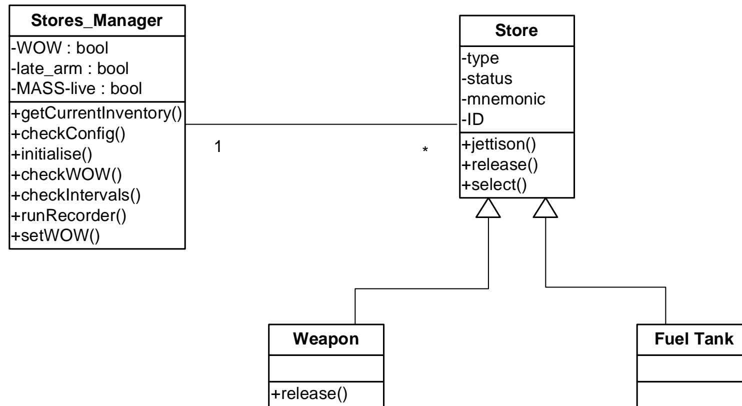


Figure 4.5: Using inheritance to create new classes

In figure 4.5, two new subclasses of class Store have been created using inheritance. These classes are Weapon and Fuel Tank, which are both types of store that may be used on an aircraft. Both these subclasses will inherit the attributes and operations of the store class. They may also add their own attributes and operations as required, but for simplicity such changes are not shown in figure 4.5. The weapon class has however also redeclared (provided a new implementation of) the inherited *release* operation. This new release operation may require different pre and postconditions to the original. Polymorphism allows store to become attached to instances of weapon. When a stores manager object makes a call to the release operation, then dynamic binding will ensure that the redeclared version of release (in the weapon class) is called, rather than the original version in store. The problem here is that stores manager only has visibility of the contract for release in *store*, and it is possible for weapon to violate this contract. This could occur in two ways:

- If weapon makes the precondition of release stronger than it was in store, then some calls which are correct from the store manager's viewpoint will not be handled properly.
- If weapon makes the postcondition of release weaker, then the results promised by store will not be achieved.

If the contracts under consideration were safety contracts, then these violations could be potentially hazardous. The reverse of these changes (weakening the preconditions, and strengthening

the postconditions) is however permitted. If this rule is followed then LSP will be complied with. This thesis proposes that this rule can be used to help in assesing whether a subclass is a "safe subclass" or not. It should be noted that a subclass of a class for which a safety contract exists is not *necessarily* a safe subclass just because LSP has been complied with. The safety of the subclass must be considered in terms of how the behaviour of the subclass may contribute to the system level hazard. The safety contract put in place for the subclass must always define safety obligations sufficient to ensure that the subclass does not contribute to a system hazard. The safety contract placed on that subclass must *also* comply with LSP. This is illustrated more clearly in the examples below.

Some simple examples of how this rule can be useful in creating safe subclasses are now considered. The simplest case is when the operations are inherited directly from the parent class. In this case subclasses can inherit the safety contracts directly from the parent class. In other cases, additional functionality can be added by a subclass, by introducing a new operation. For example the weapon class may introduce a detonate() operation which is specific to weapons, and not a general function of a store. In this case, in the same way as when a new class was introduced to the class design, the operation must be analysed to identify if a safety contract is required.

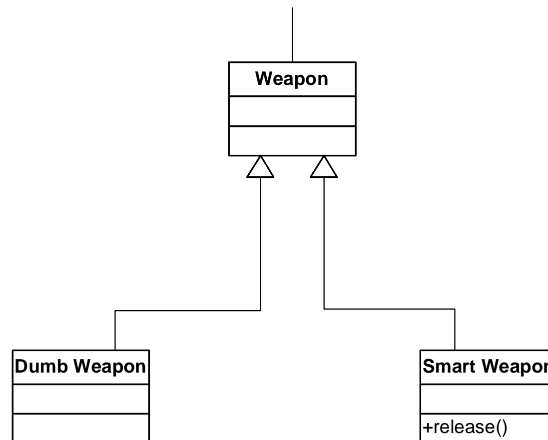


Figure 4.6: Operation redefinition using inheritance

A subclass may also redefine an operation inherited from a parent. In figure 4.6, the inheritance mechanism has been used to introduce two different types of weapon into the system, dumb weapons and smart weapons. Although both are types of weapon, inheriting attributes and operations of the weapon class, the two types of weapon also have certain features which are different. In this case, the way in which a smart weapon is released differs from other types of

weapon, so the release operation has been redefined for the smart weapon. Analysis of this new release operation may reveal that the safety contract required is different from that inherited from the parent. It must be ensured that the new safety contract follows the rules described earlier for ensuring that LSP is complied with.

Let us assume, for example, that analysis of the redefined release operation has revealed that the release must be quicker for a smart weapon to ensure that a HSFM does not occur. The safety contract for the release operation must therefore be changed from that which was inherited, in order to reflect this. This involves strengthening the postcondition of the release operation to specify that the deadline is reduced from 50ms to 30ms as shown below, the rest of the safety contract is the same as that defined for the release operation of store previously.

```
context Smart Weapon::Release()
pre:  previousRelease.at+100<=Time.now
post:  let message:OclMessage=stores_manager^checkWOW() in
message.hasReturned()
and
message.result()=false
and
Time.now<=Time.now@pre+30
```

Since the postcondition has been strengthened, this satisfies the rules established earlier for complying with LSP. This safety contract is therefore acceptable, and the safety obligations from this contract must be met by the relevant objects.

Let us now consider that the analysis also revealed that, to be safe, the release of a smart weapon must occur later after a previous release than was specified for the store class. This would require that the precondition in the safety contract of store be strengthened for a smart weapon. This breaks the rules established earlier complying with LSP, so this is not an acceptable safe subclass. In such a case, the only way to avoid violating these rules would be to strengthen the precondition of the safety contract of the parent class as well. This would mean that the safety contract of store would need to be changed to reflect this, as well as the safety contracts of all the other sub-classes of store. This in turn would result in changes to the safety obligations of many objects in the system, which would become unnecessarily stringent.

The result of this is that objects must be verified against the new obligations and may no longer

meet safety obligations where previously they did. This situation is obviously to be avoided where possible. It is therefore desirable that when safety contracts are specified, they are made as flexible as possible. That is that they are as weak as they can be whilst still defining safe behaviour. If this is done, then through inheriting the safety contract, much analysis effort can be saved when introducing classes to the system design. Another approach to introducing new elements to a system is to reuse existing elements. This is discussed in the next section.

4.4.2 Supporting the Reuse of Design Elements through Safety Contracts

As discussed in chapter 2, encapsulation is a feature of OO systems that provides a great potential for reusing elements of one system in another similar system. If software classes providing the required functionality already exist, then encapsulation ensures that including them in a new system is relatively straight forward. The advantages of this are that the part of the system being reused does not need to be developed from scratch, considerably saving on effort and hence cost. However, as was discussed for design changes previously, in safety critical applications it must be ensured that reused design elements will not affect the safety of the system. It is desirable that assuring safety can be achieved in a way which will minimise the effort required. Once again, safety contracts can be used to facilitate this. The part of the design which is to be reused must behave safely as part of the system in which it is to be used. This means that the reused element's behaviour must not contribute to any HSFMs. Since safety contracts are in place within the system design, the safety obligations relevant to the elements to be reused can be clearly identified. When reusing classes in the new system it is therefore not sufficient that they provide the functionality required, it is also necessary that they meet the safety obligations identified from the contracts. If the elements to be reused can fulfill these obligations then then they can be safely used as part of the system. It is important to note that the safety obligations are specific to the the system for which they were derived. This means that the safety obligations of a reused element from another system cannot be simply carried across into the new system. Instead it is necessary always to demonstrate that the reused element meets the safety obligations imposed upon it by the system in which it is to be used. Where the reused element is from a very similar system, which placed similar safety obligations on that element, it would be relatively easy to demonstrate compliance with the safety obligations in the new system, however it is necessary that this is done explicitly.

It is only through identifying explicit safety obligations on classes in the system design that it

is possible to know if an existing element is safe to use in that system. If this were not possible then the system would need to be reanalysed as a result of the change, and thus the benefits of reuse would not be realised.

4.5 Conclusions

In this chapter, the concept of safety contracts was introduced as an ideal way to represent DSRs as part of an OO design. It has been shown how such safety contracts can be constructed based on the analysis that was described in chapter 3. It has also been shown how OCL provides an implementation independent notation for specifying contracts, and has the expressive power necessary for representing the constraints required in a safety contract. These safety contracts can be then be used to identify the safety obligations relating to objects in the system. It has been shown how the specification of safety contracts makes it easier to deal with changes to an OO design, and the reuse of design elements, whilst ensuring that the system under consideration remains safe. In this way a safety contract approach supports maintainability and reuse, key benefits of adopting an OO approach, for safety critical OO systems.

By showing that the safety obligations arising from safety contracts have been met, evidence as to the safety of the system is generated. This evidence can be used in demonstrating that the OO system is safe to operate. In order to demonstrate this clearly, it is necessary to produce a safety argument for the system. In the next chapter, the way in which a safety argument can be generated, to demonstrate that an OO system, developed using the approach described in chapters 3 and 4, is safe to operate, is investigated. It shall be seen how the development of safety contracts assists in establishing an effective safety argument structure. Producing a successful safety argument is a key aspect of certifying an OO system such that it may be used in safety critical applications.

Chapter 5

Creating a Safety Argument for OO Systems

5.1 Introduction

In chapter 2 it was seen that in order to certify a safety related system it is necessary to produce a safety case for that system. A key part of this is providing a clear and defensible safety argument that the system is acceptably safe to operate within a particular context. For the approach described in this thesis to be used successfully in safety related systems, it is necessary that a robust safety argument can be produced, which shows how the process ensures the resulting software system is safe. Chapter 3 described a process for analysing OO systems for safety, then in chapter 4 the output of the analysis was used to define safety requirements in the form of safety contracts on the system design. Once these requirements for the system have been defined, the system is verified against these requirements to generate evidence that the requirements have been met. This evidence will form part of the safety argument. In this chapter an appropriate structure for such an argument is developed. The structure should provide the flexibility required to support change and reuse of the system. If the argument structure does not provide this flexibility then the certification effort could easily negate the advantages gained elsewhere in the process. Safety case patterns are developed which can be used to aid the development of safety arguments for OO systems.

5.2 Modular Safety Argument Structures

The traditional approach to producing a safety argument for a system is to produce a monolithic safety argument for the entire system under consideration. This means that the entire system is considered as one entity. There is nothing inherently wrong about this approach, and acceptable monolithic safety arguments have been produced for many complex systems. Indeed it would be possible in principle to produce a monolithic argument for an OO system, which successfully made an argument that the system was safe to operate. In this section, however, it will be shown that using a monolithic structure for the safety argument has certain drawbacks and, instead, an alternative modular approach is more appropriate for OO systems.

In chapter 2 GSN was introduced as an effective notation for representing safety case arguments. In this chapter GSN shall be used to represent the safety argument structures. Initially different modular argument structures will be considered. These can be represented using the *module* and *away goal* extensions to GSN [30]. A module is a self-contained component of the argument and supporting evidence relating to a particular aspect of the safety case for the system. Figure 5.1 shows a modular representation of the monolithic safety argument. As such only one argument module is required. This module contains the safety argument and evidence relating to all of the classes in the whole of the system under consideration.

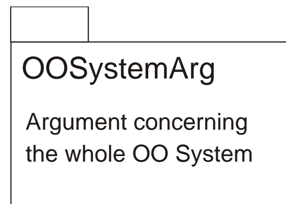


Figure 5.1: Monolithic safety argument structure

In assessing whether a particular argument structure is good for an OO system, the criteria that shall be used are the same as have been considered throughout this thesis. That is that the argument structure is robust in the face of changes to the system design, and that the structure supports the reuse of design elements. Therefore, the amount of the argument that is affected by a change to a system design should be as small as possible. A change to a design will effect a module if the argument within that module relates to the changed part of the design in some way. It should also be possible to identify and isolate modules of the argument relating to specific parts of the system design, such that if that part of the system were reused in another system design, the corresponding module may also be reused as part of the safety argument for the new system. This will reduce the effort required, compared with developing the relevant

argument module from scratch.

When considering the monolithic safety argument structure, there is found to be very poor support for change. Since there is just one argument for the whole system, the argument contained within the module will commonly (and almost inevitably) contain a large web of argument dependencies between different elements of the argument. Although the argument produced could be valid, any change to the design of the system would affect large parts of the argument. This would mean that in order to remain valid for the changed design, the whole argument would, in effect, need to be revisited. Since the arguments produced can be extremely large, this could represent a huge effort. In addition, it could be potentially very difficult to identify the parts of the argument that have been affected by the change. Due to the monolithic structure, the effects of the change could be spread throughout the argument, making it difficult to identify the affected parts.

If part of the design were reused in another system then it would be extremely difficult, with this argument structure, to reuse the corresponding part of the safety argument in the argument for the new system. The argument is very highly coupled, and therefore separating out the argument relating to the particular element of the system being reused, without losing important elements of that argument, would be very difficult. It is for these reasons that it is felt that this traditional approach to constructing a safety argument is insufficient to retain the desirable features of OO designs. An alternative argument structure is therefore required.

Figure 5.2 shows another possible argument structure that might be used. In this structure, the top-level argument module provides the scope of the overall system argument, and the child modules present independent safety arguments about the classes in the system design. The links between the modules indicate that a claim in one module is solved by the argument contained in another module. These links are made using away goals as illustrated in section 5.3. Evaluating this argument structure in a similar manner to the monolithic structure reveals some interesting observations. It can be seen that this structure has better support for change than the monolithic structure. Because there is a separate module containing the argument relating to the safety of each class, it is easier to identify the impact of design change on the argument structure. For those classes which have been changed in the design, the relevant class argument module will also have been affected, and must be updated to reflect the change. For those classes that have not been changed, the corresponding module will remain unaffected. Due to the separation in the argument there are no knock-on effects throughout the argument structure.

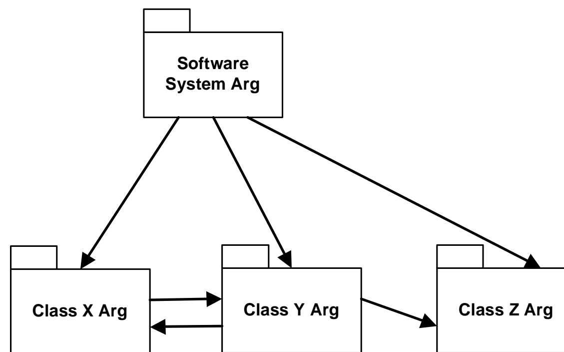


Figure 5.2: Modular argument structure including class modules

Another type of change that would commonly be encountered is to introduce a new class into a system design. At first it would seem that the argument structure in figure 5.2 supports this type of change, as a new argument module relating to that new class could be added to the structure. In fact, things are more complicated than that, and this structure is far from ideal. It has been seen throughout this thesis how the interactions between class objects in the system are crucial to the behaviour, and hence the safety of the system. In the structure in figure 5.2, the argument about the safety of the interaction between the classes is contained within the class argument modules themselves. When a new class is added, it may interact with a large number of the other classes in the system. Therefore the addition of a new class will also affect the argument modules of all the other classes with which that class interacts.

It would similarly appear that there is a greater reuse potential with this argument structure. It is now easier than in the monolithic case to identify the relevant part of the argument to be reused. However, due to the dependencies that exist between modules, resulting from the interactions, the reuse potential of the argument modules is also limited. When an argument module is reused the interactions will change from those with other classes in the old system to those in the new. This may require a great deal of reworking of the argument module for it to be valid in the new system. This could negate the advantage gained from reusing the argument module.

The problems associated with the previous argument structure can be dealt with by introducing a module which deals explicitly with the safety argument about the interactions between classes. This argument structure can be seen in figure 5.3. In this structure, an argument about *all* the interactions is made in a separate module. There are no longer any links between the class argument modules as the class arguments no longer have responsibility to reason about their effect on other classes. The individual class argument modules now just make an argument

that they will meet any constraints arising on them from these interactions. As a result of this, handling changes to the system design is now much easier. For example, introducing a new class to the system design, in addition to a new argument module for the new class will primarily impact just the interactions module. This will be changed to handle the new interactions introduced. There is no longer a knock-on effect on the rest of the argument, as there was with the other argument structures.

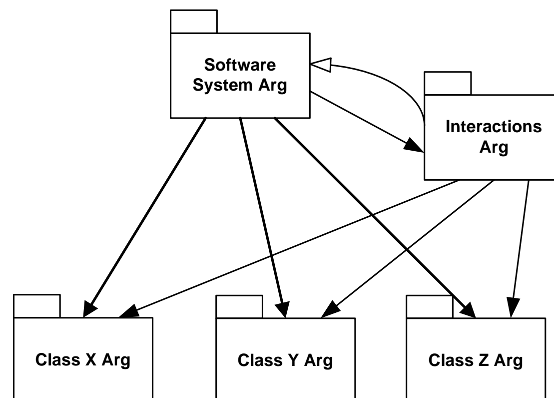


Figure 5.3: Modular argument structure with separate interactions argument

The structure also provides better potential for reusing parts of the argument. As stated earlier, because of the interactions module, the class modules no longer need to reason about other classes in the system as part of their argument. This means that any of the class argument modules could be used as part of the argument for a different system with minimal rework required. It is for these reasons that the argument structure in figure 5.3 is proposed as the most desirable for presenting a safety argument for an OO system. Later in this chapter, the effects of change and reuse on the safety argument itself will be considered in more detail. Firstly it is necessary to develop those arguments, which are contained within each of the argument modules. In the next section patterns for the arguments are presented in GSN. These patterns can be instantiated for the OO system to which they are applied.

5.3 Developing the Safety Arguments

In this section, the safety argument necessary to show that an OO software system is acceptably safe is developed. The argument will show how the safety of the system is demonstrated based upon the approach described in chapters 3 and 4 of this thesis. The argument shows how the evidence generated using this approach supports the top-level claims about the safety of the system. The argument will be developed using the argument structure suggested in figure

5.3. Each of the modules shall be developed, with the away goals which link the arguments in different modules highlighted where necessary. Firstly the software system level argument module is developed.

5.3.1 Software System Level Argument

The software system level argument module provides the top-level argument that the software is acceptably safe. The safety argument pattern for this module is shown in Appendix A. The argument pattern is described in the format proposed by Kelly in [29].

The way in which the safety of the software is demonstrated is through a hazard control approach. This reflects the process described in chapter 3 which takes a top-down approach starting from the system level hazards and identifying how the software may contribute to these hazards. The top level claim is made in the context of the description of the current system. The strategy adopted is to argue over all the identified hazards. The context for this will be a hazard log, or other repository of known hazards for the system. The argument then shows that the software contribution to each system level hazard is acceptable. This is done in the context of the software design (this may take the form of a UML model, or other design description), and also in the context of the HSFMs identified for each of the hazards. This argument is made by showing that all the HSFMs have been identified correctly (G3_Sys), and will not occur in, or are mitigated by, the software (G4_Sys).

In order to show that the HSFMs do not occur, the strategy adopted (again based on the approach proposed in this thesis) is to argue separately that the interactions are safe and that the individual classes themselves will not behave in such a way that they will contribute to the HSFMs. This strategy also fits in with the prefered safety argument module structure in figure 5.3. It is also necessary to argue that there are no unintended interactions between classes in the system. The argument as to why unintended interactions do not occur is specific to properties of the software architecture used in implementing the software. The argument supporting this claim has therefore been left undeveloped as this lies outside the scope defined for this thesis. The arguments could, for example, appeal to the rigour of a partitioning mechanism which ensures objects do not interfere. There are two away goals as part of this argument. These are goals which are addressed by arguments made in other modules (in this case the interactions argument and the argument for the class).

5.3.2 Interactions Argument

The Interactions Argument Module Pattern can be seen in Appendix B.

This argument module argues that the interactions that occur between classes in the system are acceptably safe. This argument satisfies an away goal from the system-level argument. It was seen in chapter 4 how DSRs can be specified in the form of safety contracts, from which safety obligations for classes in the system can be identified. This approach is reflected in this argument. This is done by arguing firstly that the contracts have been correctly identified (G2_Int), and then that the safety obligations arising from those contracts have been satisfied (away goal G2_Class). It is shown that the contracts are identified correctly by arguing that the relevant interactions for each of the identified HSFMs have been identified (G5_Int). This is done by appealing to evidence generated from the fault tree and interactions analysis described in chapter 3. This claim is made under the assumption that the software design contains complete and accurate information on the interactions in the system. Without this the analysis would not be correct. It is then shown that for each of these relevant interactions, the necessary DSRs have been identified to ensure that the HSFM will not occur (G6_Int). These DSRs are captured as safety contracts for each relevant interaction. This is again argued by appealing to various forms of evidence which are generated from the analysis process described in chapter 3. This includes evidence from SHARD-style analysis, timing analysis of interaction diagrams and analysis of statecharts. The resulting contracts are captured as context for the claim G7_Int.

The argument that all the classes can meet the safety obligations arising from the contracts is made through away goal G2_Class. This is a goal from the class argument module. By making this argument in the class module, it is ensured that the argument about the behaviour of the classes is separated from the argument made about the interactions. Again, this strategy fits in with the argument module structure in figure 5.3.

5.3.3 Class Argument

An argument module is developed for each class in the system design. The argument developed for each class is as shown in the Class Argument Module Pattern in Appendix C.

The purpose of the class argument is to show that the class will not contribute to any of the relevant HSFMs. This is done firstly by arguing that the safety obligations on each instance of that class (each object) arising from its contractual constraints have been satisfied (G2_class). This argument is referenced as an away goal from the interactions argument.

G2_Class is made in the context of the safety contracts derived as part of the interactions argument. The argument involves identifying all the contractual obligations from the interactions in which that object is involved. Chapter 4 of this thesis described how such safety obligations are identified. It is necessary then to show that those obligations are met, chapter 4 discussed ways in which the software may be verified against its safety obligations. The evidence generated from verification activities is captured through solution S1_Class. This solution represents a placeholder for the evidence used to support claim G5_Class. Once a specific verification strategy is decided upon, S1_Class must reflect how the verification evidence supports the claim. This may require further decomposition of the argument below G5_Class.

Secondly, it is necessary to argue that the class itself will not behave in such a manner as to contribute to the HSFM (G3_Class). This is to cover behaviour which classes may exhibit other than as a result of interactions with other classes. This will be, mainly, such things as subtle value failures which could be present in an implemented system. Goal G3_Class provides this argument. This goal requires further development to show how such contributions are identified.

5.4 Handling Change and Reuse

It has been shown in the previous section how a safety argument for an OO system can be produced using the approach described in this thesis. A number of claims were also made about the argument structure used. It was claimed that the argument is robust to changes that may be made to the system design. It was also claimed that elements of the argument for a design may be reused, when elements of that design are used in other systems. In this section a safety argument made for the SMS using the patterns presented in section 5.3 shall be considered. A number of changes that might occur to the SMS design are considered, and the effects upon the SMS safety argument resulting from these are examined in some detail. Similarly, a reuse scenario will also be considered, to illustrate how safety argument elements may be reused.

5.4.1 Changes to the Design of a Class

The change to a class design that will be considered is a change to the algorithm for the checkConfig() operation (see figure 3.4) of an object of type stores_Manager. This operation checks the configuration of the stores on the aircraft to identify which store is attached to which station. This operation could contribute to HSFMs associated with the release of incorrect store

hazard. Safety obligations identified for the stores_Manager object must still be met when the change has been made. It must therefore be shown that all safety obligations, specifically those associated with checkConfig() are still met.

The safety argument must be updated to reflect the changes that have occurred. This will ensure that the argument remains valid. The effects of the change on the arguments in each of the argument modules will be considered. In the software system level argument, the only claim which is affected by a change to an object such as this, is the goal G1_Class, which considers the contribution of the the class to the HSFMs. This goal however is an away goal in the class argument module, and is therefore not dealt with in the software system argument. The software level argument module is therefore unaffected by this change.

Investigating the interactions argument, it can be seen that this module of the argument is also unaffected by this change. The interactions argument deals with the identification of the contractual constraints necessary to mitigate against a HSFM. This module will therefore be affected by design changes which lead to these constraints requiring alteration. This would be due to changes to the nature of the interactions between the classes. Since these interactions have been unaffected by this change, the DSRs and hence safety obligations remain unchanged. Goal G2_Class deals with the satisfaction of the safety obligations by each of the classes, but again this is an away goal in the class argument module.

It is therefore only the class argument module for the stores_Manager class (the changed class in this example) which will be affected by the change. The most obvious effect on the argument is to goal G5_Class. It must be shown that the safety obligations for the stores_Manager object can still be satisfied with the algorithm for checkConfig() having been changed. This will involve generating evidence that the obligations are met by the stores_Manger object. This evidence is recorded as part of S1_Class. By ensuring these parts of the argument are updated to reflect the change, the argument will remain valid. Most of the argument structure, as was seen here, is unaffected by this type of change.

5.4.2 Introducing a New Class

A more complicated type of change is to introduce a new class into the system design. The introduction of a new store class into the SMS design shall now be considered. This change will have more effect on the argument structure than the change discussed previously, as it will result in changes to the interactions that occur between objects. Starting again by looking at the system level argument module, this time goal G1_Int will be affected, as it must be shown

that interactions introduced by the new store class are acceptably safe. This is an away goal in the interactions argument module. This module will now be considered.

It must be shown, as part of the interactions argument that all contracts have been correctly identified. As part of this, G5_Int requires that all interactions relevant to each HSFM are identified. To ensure that this remains valid when a new store class is introduced, it must be identified if any of the interactions introduced between the new class, and existing classes in the system, could contribute to a HSFM. The results of the analysis performed to determine this is captured in the solutions S1_Int and S2_Int.

Once the argument is updated to reflect this new analysis, the new interactions identified as contributing to a HSFM must have DSRs defined. The analysis used to derive the DSRs is captured in the relevant solution elements. These additional DSRs are identified in safety contracts, and the context C4_Int is updated to reflect the changes to the safety contracts. The interactions argument has now been updated to reflect the addition of the new store class to the system design. The changes made to the safety contracts results in new safety obligations on classes in the system. It must be checked that the objects can meet these obligations. This will impact the class argument modules in just the same manner as with the previous change. That is that more evidence is needed to show that the obligations are met. It can be seen with this simple example that most of the effect of introducing a new class is on the interactions argument module. The effect on the class argument is minimal. Containing most of the impact within one module is an advantage of the chosen argument structure.

5.4.3 Reusing a Class

It may be the case that a new class being introduced to the SMS design is one that has previously been developed for another similar system. For example, a new type of weapon may have been used previously on a different aircraft. If this is the case then as well as reusing that part of an existing system design, there is much benefit to be gained from also using as much as possible of the corresponding safety argument. So if an argument module exists for the new weapon type, this module could be included in the argument structure for the SMS. The key thing in doing this is that the links that exist between the reused argument module and the other modules in the argument structure remain valid. In the case of the class argument module, the crucial link is the contextual link made to C4_Int in the interactions argument module. It is the safety contracts that are defined in the interactions argument which define the set of safety obligations for the class. These contracts will be specific to each system, and therefore the safety

obligations which must be satisfied will also be specific. Once these obligations are identified from the interactions argument, evidence must be generated to show that the obligations are met. It is possible that, for example, testing evidence which formed part of the class argument in the previous system, may also be applicable in discharging obligations in the new system. Such evidence may be reused.

This illustrates how separating out the argument about the individual classes from the argument about the interactions makes it much more possible to reuse large parts of the class arguments, thus reducing the rework required. This is another indication of the advantage of the argument structure from figure 5.3.

5.5 Conclusions

In this chapter safety argument patterns have been presented for use in creating arguments about the safety of OO software system designs. Such safety arguments are a crucial part of the safety case for a system. The safety contracts resulting from the analysis process described in this thesis can be used to generate the evidence which is required in support of the safety claims which must be made about the system. It has been seen in the discussions in this chapter that certain goals in the argument modules have not been fully developed, specifically G7_Sys, G3_Class and G5_Class. The way in which these claims are supported is outside of the scope of the method developed in this thesis. In order to be a complete and compelling argument, the claims and evidence supporting the undeveloped goals must be provided. Although this has resulted in incomplete argument patterns where goals are outside of the scope of the thesis and have been left undeveloped, it has been suggested in the discussions in section 5.3 how the claims may be supported. With these goals fully developed, the arguments produced using these safety argument patterns can be used to show how this evidence demonstrates that the resulting software system is acceptably safe to operate. The argument presented has been split into modules. The module structure has been chosen such that alterations required to the safety argument are minimised should there be changes made to the system design. The chosen modular structure also maximises the amount of the argument that can be reused when elements of the design are used in other systems. It is the use of a safety contract approach which allows such a modular argument structure to be used, as the argument can thus be separated into modules regarding the interactions, and the individual classes.

The SMS has again been used as an example, to illustrate the effects that changes to a design,

or reuse of a class, would have on the safety argument. Producing and maintaining a safety argument for a system are typically costly and time consuming activities. Therefore reducing the effort required in maintaining the validity of the safety argument is of great advantage. Since benefits are often claimed for an OO approach due to their being maintainable and reusable, it is particularly important for OO systems that a safety argument can support such properties.

Chapter 6

Aircraft Avionics Control System; A Case Study

6.1 Introduction

This chapter presents the results of a case study undertaken upon an avionics control system design produced by a large aerospace company. The purpose of the case study is first and foremost to show how the analysis process described in chapter 3 of this thesis can be used to generate DSRs upon a large-scale OO software design in a systematic and rigourous manner. The case study will also be used to illustrate how these DSRs can be used to define safety contracts upon the design, and how a safety argument can thus be constructed for the system. All figures referred to in this chapter may be found in appendix D.

6.2 System Overview

This description of the system and all other information presented about the system and the system design, including all design diagrams are taken directly from the Software Requirements Specification (SRS) document [77], and the Software Design Description (SDD) document [76]. Some of the information used may have been altered where required for commercial reasons.

The Avionics Control System (ACS) is the main navigation system of an aircraft, and can be considered to have two main roles:

Mission Management Management of the flight paths and mission objectives.

Navigation Calculation of aircraft navigation status from sensor data.

The system architecture diagram in figure D.1 shows the logical architecture of the system. The functionality of the system is described below.

> An ACS continuously collects sensor data to estimate the actual state of an aircraft, compute desired aircraft state with respect to guidance modes, and performs actions that advise pilots and directly manipulate aircraft effectors in ways that bring actual and desired state in closer agreement. The system modelled is a simplified ACS which addresses a small subset of the functionality normally associated with an ACS used by modern military aircraft.
>
> The pilot interacts with the ACS via a control panel and a VDU located in the cockpit. The ACS obtains navigation data from a suite of navigation sensors. The ACS provides mission data, flight path information and aircraft navigation status data for presentation to the pilot via the VDU. In addition it computes the required heading and altitude to maintain the programmed flight path. The system computes and displays cues on the VDU which prompt the pilot to fly the aircraft in a particular direction in order to maintain the programmed flight path. Finally, commands are provided to the autopilot which ensures that, when it is engaged, the autopilot also maintains the programmed flight path.

6.3 Safety Analysis of ACS

The safety analysis of the ACS software design will follow the process steps for analysis of the functional, temporal and value aspects of the system as described in chapter 3 of this thesis. The relevant design artifacts for each stage of the analysis are introduced as required. Firstly the results of the functional analysis are described.

6.3.1 Functional Analysis

6.3.1.1 Step one: Identify Hazards

The system level PHI activities identified just one hazard associated with the ACS. This hazard is stated as follows:

Hazard Aircraft descends below minimum safe altitude during low level operations.

This is a hazard for the aircraft as it could lead to loss of the aircraft due to impact with the ground.

6.3.1.2 Step two: Define hazardous failure modes

This step of the analysis identifies potential failure modes for the ACS software, which may lead to the identified hazard. These are referred to as Hazardous Software Failure Modes (HSFMs).

During low level operations the system uses the terrain following (TF) function of the ACS to ensure that a minimum altitude selected by the pilot is maintained. System level hazard analysis identified the following two HSFMs associated with the system hazard:

HSFM 1 With TF enabled, the ACS fails to warn the pilot that the aircraft has descended below the minimum safe altitude.

HSFM 2 With TF enabled, the ACS fails to command the autopilot to gain height when the aircraft descends below the minimum safe altitude.

It will actually be seen that these two HSFMs are essentially the same, with the difference being whether the TF output is sent to the pilot or the autopilot. It can also be argued that, provided the pilot is warned when the aircraft descends below the minimum safe height, the autopilot can be disengaged, and avoiding action taken. For these reasons, just HSFM 1 shall be considered for this case study. For completeness it would also be necessary to perform analysis of HSFM 2, however in this case, it will be highlighted where the results for HSFM 2 would be different from those obtained for HSFM 1.

6.3.1.3 Step three: Identify interaction failures

This step identifies failures in the software which may lead to the identified HSFM. This involves investigating the sequence of interactions that could result in the HSFM. A fault tree can be constructed for the HSFM using the sequence diagram for the relevant scenario. There is one use case related to the HSFM which is 'maintain minimum height'. The use case is given in figure D.2. The normal scenario for this use case is described in the sequence diagram in figure D.3. In this scenario, the TF mode is selected by the pilot. Once TF mode is enabled, the pilot is able to select a minimum height for low level operations. This minimum height is read by the ACS software and displayed back to the pilot via the VDU. The ACS continually monitors the aircraft navigation sensors to determine the aircraft's current height. The set of sensors

that are used for doing this is dependant upon the current navigation mode of the ACS. The ACS uses the current aircraft height to determine if the minimum height has been maintained. A TF cue is calculated to maintain the minimum height. This TF cue can be displayed to the pilot, or if the autopilot is in use, can be sent as a vertical command to the autopilot.

The fault tree is constructed by working back through the sequence of interactions that occur, as described in the sequence diagram in figure D.3. The top level event in the fault tree is 'ACS fails to warn pilot when aircraft altitude falls below minimum safe altitude'. The fault tree is shown in figure D.4.

The interaction failures can be identified from the set of basic events in the fault tree. There are 11 basic events as listed below:

1. Failure of Flight_Director to decode minimum height
2. Failure of Minimum_Height() - *Interaction*
3. Failure of Read minimum height signal - *Interaction*
4. Pilot selects wrong minimum height value via control panel
5. Failure of Navigator to calculate current height correctly
6. Failure of Sensor interactions - *Interaction*[1]
7. Failure of Sensor
8. Failure of Update_Height() - *Interaction*
9. Failure of Flight_Director to calculate radar cue correctly
10. Failure of Display_TF_Radar_Cue() to display - *Interaction*
11. Failure of Display TF radar cue signal to VDU - *Interaction*

Those events which are interaction failures are indicated above and will be investigated further in the next step of the analysis. The other basic events are different types of failure which will be dealt with in other parts of the analysis.[2]

[1] It should be noted that there are a number of different sensors and sensor interactions, as can be seen in figure D.3. The set of sensors used depends uppon the current navigation mode. For simplicity, and for presentational reasons, the sensors and their interactions are each identified as just one failure event. It should be evident to the reader that the analysis for all sensor interactions can be generalised in this manner.

[2] Basic events 1, 5, and 9 are dealt with in the value analysis. Event 4 is a human factor failure relating to the behaviour of the pilot. This failure is outside the scope of this process. Event 7 is a hardware failure, which again falls outside the scope of this process.

6.3.1.4 Step four: Investigate causes of failures

This step of the analysis involves determining potential causes of the failures identified in step 3. There are two elements to this step of the analysis. Where statechart representations of the relevant objects in the system design are available, these statecharts can be analysed to understand the causes of hazardous failures. However the analysis will first focus on SHARD-style analysis of the interactions.

6.3.1.5 Step 4a: SHARD-style analysis

The interactions to be investigated for failures, as identified at step 3 are:

1. Min_Height()
2. Read minimum height signal
3. Sensor interactions (Get_Height(), Get_Position(), Ground_Level())
4. Update_Height()
5. Display_TF_Radar_Cue()
6. Display TF radar cue signal

The details of the interactions are obtained from the class diagrams (parameters, return variables etc.) and sequence diagrams (client and supplier objects). These interactions are analysed using the SHARD-style analysis. This investigates deviations from the expected behaviour of the interactions, in order to identify hazardous behaviour. The results of the analysis are shown in figures D.5 to D.10.

6.3.1.6 Step 4b: Statechart analysis

Where a statechart representation of an object is available, this statechart should be analysed to give further understanding of potential causes of hazardous failures. A simple statechart has been developed for the Navigator class as shown in figure D.11. This shows that the navigator object has responsibility for reading the sensors. The sensors that are read depends on the current navigation mode which is selected. The navigator then updates the height based on these obtained sensor readings. The failure events that are being investigated using this analysis are therefore:

Failure 6 Failure of sensor interactions

Failure 8 Failure to update height

With respect to the statechart for the navigator class, these failures can be defined as:

- Object fails to read all active sensors
- Object fails to update height

Firstly the statechart is examined to ensure that these failures won't occur in the proposed design for the statechart (figure D.11). If we consider the object in the state 'Navigation mode selected', it can be seen that when the time t reaches the time limit $Time$, the action 'Get sensor readings' is performed and the object moves to the 'Reading sensors' state. Once all the sensors have been read the object moves to state 'Updating height'. This state is exited once the height has been updated. The time t is reset to zero, such that the sensor reading cycle will continue. It can therefore be seen that if the object behaves normally, that is as defined in the statechart, then the hazardous failures under investigation will not occur.

It is necessary to also consider the abnormal, or faulty behaviour of the object. This can be done by mutating the statechart. The mutated statechart for the navigator class is shown in figure D.12. The transitions between only the three states shown have been mutated, as the other states in the original statechart are not relevant to the failures being investigated. Each mutated transition is considered for its contribution to a hazardous failure. This information is useful in defining the hazardous object behaviour at the next step of the analysis. The way in which the mutated transitions may be hazardous is described below.

A1 Sensors are not read, and height not updated.

A3 Although the sensors are read, since the time limit is not achieved, the height value may be stale.

A4 Although the object moves to the reading sensors state, the sensors will not be read, and height not updated.

A5 Although the object moves to the reading sensors state, the sensors will not be read, and height will be updated with a stale value.

B1 Current height is not updated.

B3 Height is updated incorrectly.

B4 Although moving to updating height state, the height will not be updated.

B5 Although moving to updating height state, the height will not be updated.

C1 Although height is updated, object remains in updating height state so next sensor reading cycle will not occur.

C3 Although moving to navigation mode selected state, the height has not been updated.

C4 Height is updated, however time is not set to zero, so the next sensor reading cycle will not occur.

C5 Height is updated and sensor readings will be obtained, however object will remain in navigation mode selected state, so new height will not be updated.

6.3.1.7 Step five: Define hazardous object behaviour

Each of the deviations identified as hazardous in the analysis in both parts of step 4 represents a hazardous behaviour of an object in the system design. This step of the analysis defines these hazardous behaviours in terms of the objects responsible for them.

1.1 Flight Director fails to check minimum height using Minimum_Height() call

1.2 Control Panel returns no value in response to Minimum_Height()

1.4 Control Panel returns incorrect minimum height to Flight Director

2.1 Control Panel Hardware fails to send signal to software

2.2 Control Panel Hardware sends valid spurious signal to software

2.3 Control Panel Hardware sends incorrect minimum height signal to software

3.1 Navigator fails to get data from relevant Sensor object using Get_Data() call

3.2 Sensor object returns no data in response to Get_Data() call

3.4 Sensor object returns incorrect data to Navigator

4.1 Navigator fails to update aircraft's current height using Update_Height() call

4.2 Navigator does not provide Height_In_Feet parameter for Update_Height() call

4.4 Navigator provides incorrect Height_In_Feet parameter to Aircraft

5.1 Flight Director fails to display radar cue using Display_TF_Radar_Cue() call

5.2 Flight Director does not provide TF_Radar_Cue parameter for Display_TF_Radar_Cue() call

5.4 Flight Director provides incorrect TF_Radar_Cue parameter to Status Display

6.1 Status Display fails to send signal to VDU

6.2 Status Display sends valid spurious signal to VDU

6.3 Status Display sends incorrect Radar Cue signal to VDU

A1 Navigator fails to get sensor readings

A3 Navigator fails to meet time condition for getting sensor readings

A4 Navigator fails to get sensor readings

A5 Navigator updates height before getting sensor readings

B1 Navigator fails to update height

B3 Navigator fails to get sensor readings before updating height

B4 Navigator fails to update height

B5 Navigator fails to update height

C1 Navigator ceases to renew sensor readings

C3 Navigator fails to successfully update height

C4 Navigator ceases to renew sensor readings

6.3.2 Temporal Analysis

6.3.2.1 Step one: Split scenario into tasks

In order to understand timing effects on the behaviour of the system, it is necessary to consider sequences of interactions, referred to as tasks, performed as part of a scenario. A task is defined more fully in section 3.6.1. For the HSFMs in the ACS, the scenario that is of interest is that shown in figure D.3. This scenario can be split into the following tasks:

1. Select TF mode *Begin* - Pilot selecting TF Mode via control panel
End - TF Mode enabled

2. **Select minimum height** *Begin* - Pilot alters minimum height selection
   End - Minimum height displayed on VDU

3. **Determine aircraft height from sensors** *Begin* - Determine height from first TF sensor
   End - Flight director checked current height

4. **Compute pilot TF cue** *Begin* - System computing pilot TF cue
   End - TF radar cue displayed on VDU

6.3.2.2 Step two: Investigate the effects of timing deviations

Timing deviations are applied to the tasks identified at step 1. The results of this are shown in figure D.13.

6.3.2.3 Step three: Analyse alternative scenarios

Figure D.14 shows a representation of the normal scenario from figure D.3. Feasible alternative scenarios are obtained through deviating the normal scenario. Alternative scenarios 1 and 2 in figure D.14 represent the potentially hazardous alternative scenarios. Scenario 1 was obtained by omitting the task 'Determine current height', scenario 2 was obtained by introducing tasks concurrently with other tasks.

The way in which scenarios 1 and 2 may be hazardous is given below:

Scenario 1 The current height must be determined prior to computing a TF cue, otherwise the TF cue may be invalid.

Scenario 2 If the pilot selects a new minimum height whilst a TF cue is being computed, the TF cue must be updated first to ensure the current height is maintained. Priority must be given to the compute TF cue task. N.B. This may have a knock-on effect on task execution times.

6.3.2.4 Step four: Define timing requirements

Based upon the analysis performed in the previous steps, the nature of the timing requirements needed for the various tasks is identified (i.e. deadlines, separations etc.). The actual value of these requirements is determined using specific domain knowledge of the ACS. The values specified below indicate typical values that could be expected for this system.

Select minimum height Deadline 300ms

Determine aircraft height Deadline 300ms

Max. separation 1000ms

Compute pilot TF cue Deadline 300ms

Max. separation 1000ms

Priority

These requirements upon the tasks will be decomposed to timing requirements on individual interactions.

6.3.3 Value Analysis

6.3.3.1 Step one: Identify critical data

Firstly the critical data items for the system are identified. These can be obtained from the fault tree in figure D.4. Detailed information on the nature of the data items can be obtained from the system class diagrams. For the ACS, three critical data items are identified:

Minimum_Height attribute, Flight_Director

Height_In_Feet attribute, Navigator

TF_Radar_Cue attribute, Flight_Director

6.3.3.2 Step two: Identify manipulators

All the manipulators for the critical data are stated in figure D.15. Identifying all the manipulators for the critical data ensures that the set of constraints defined for the data takes account of all the ways in which the data may possibly be corrupted.

6.3.3.3 Step three: Define constraints for critical data

The constraints on the critical data are used to check that the critical data is, and remains, valid. A safe default value for the data is also defined.

Minimum_Height Minimum_Height = normalisation_factor * analogue voltage input

and 0 < Minimum_Height ≤ min_height_threshold

else Minimum_Height = safe_default

Height_in_Feet Height_in_Feet from valid sensors should not differ by >5%

and rate of change should not exceed ± rate_of_change_threshold

else Height_in_Feet = safe_default

TF_Radar_Cue -100 ≤ TF_Radar_Cue ≤ 100

else TF_Radar_Cue = safe_default

6.4 Defining Safety Contracts for the System

The hazardous behaviour identified through the analysis performed in section 6.3 is used to specify Derived Safety Requirements (DSRs) on the system design in the form of safety contracts. The safety contracts that are required for the ACS system are given below:

```
context Control_Panel::Minimum_Height() : Height_in_Feet
pre:
post:result=voltage_input * 200
and
0<result<=1000
and
Time.now<=Time.now@pre+100
otherwise: result=500


context INS::Get_Height() : Height_in_Feet
pre: previousGet_Height.at+1000>=Time.now
otherwise: self.status=invalid
post:previousGet_Height.result-result<=8000
and
Time.now<=Time.now@pre+50
otherwise: self.status=invalid


context RADALT::Get_Height() : Height_in_Feet
pre: previousGet_Height.at+1000>=Time.now
otherwise: self.status=invalid
post:previousGet_Height.result-result<=8000
```

```
and
Time.now<=Time.now@pre+50
otherwise: self.status=invalid


context DMG::Get_Height() : Height_in_Feet
pre: previousGet_Height.at+1000>=Time.now
otherwise: self.status=invalid
post:previousGet_Height.result-result<=8000
and
Time.now<=Time.now@pre+50
otherwise: self.status=invalid


context Aircraft::Update_Height(Height_in_Feet : Integer)
pre: Height_in_Feet>0
post:Time.now<=Time.now@pre+50


context Status_Display::Display_TF_Radar_Cue(TF_Radar_Cue)
pre: previousDisplay.at+1000>=Time.now
and
-100<=TF_Radar_Cue<=100
otherwise:TF_Radar_Cue=100
post:Time.now<=Time.now@pre+100
```

6.4.1 Identify Safety Obligations

The safety obligations for objects of each class can be identified as a set of pre and post condition requirements. These obligations can be captured in tabular form as shown in figures D.16 to D.21.[3] Note that the real value of the tables is only seen where classes have multiple obligations arising from contracts on different interactions (such as for the Navigator class). However, for consistency and completeness, tables have been developed for all classes with any safety obligations.

[3] Of the three sensor classes considered in the analysis, a table of obligations has only been produced here for the INS. The safety obligations tables for RADALT and DMG will be identical to that for the INS.

6.5 Creating a Safety Argument for the ACS

The analysis performed as part of this case study can be used in the construction of a safety case for the ACS software system. Figures D.22 to D.24 show the argument represented in GSN using the patterns developed in chapter 5. Only the class argument for the Navigator class has been developed here, however similar arguments for the other classes in the system design can be produced using the same class argument module pattern.

6.6 Conclusions

In this chapter, a case study has been used to demonstrate how the analysis techniques developed in this thesis can be successfully applied to a software system design. Once the fault tree had been developed, the functional part of the analysis was straight-forward using the deviation patterns for SHARD and statecharts analysis proposed in this thesis. Working systematically through the basic events in the fault tree ensures that all the failures that could lead to the hazard under investigation are analysed. The case study highlighted the importance of taking a decompositional approach, to ensure the fault tree is developed effectively. This means that the failures should first be considered at the level of the tasks involved in the scenario (eg. minimum height selection incorrect), before working back through the sequence of interactions in that task, starting with failure of the output of the final interaction. By constructing the fault tree in this manner, it was found that the basic events can be extracted fairly simply.

The case study has demonstrated that performing the timing analysis at the level of tasks allows a set of necessary requirements to be identified without having to perform analysis of individual interactions, which would be much more complicated and time-consuming. This approach also has the advantage of allowing the system designer to have some freedom in assigning timing constraints to specific operations. The case study has highlighted the fact that defining values for timing requirements requires a great deal of system knowledge. Therefore the analysis performed is focussed on identifying the *nature* of the timing requirements, rather than their value. Similarly for the value aspects of the analysis. The important benefit of the value analysis in the ACS case study was seen to be ensuring that the critical data, and the manipulating interactions were identified correctly. The system designer again has a crucial role in assigning the actual constraints upon the data items.

One of the key results of the case study is that it demonstrated how effective the output of the analysis is for defining safety contracts upon the software design. The process used was seen

to provide a systematic way of decomposing safety properties of the system down to individual requirements upon the objects in the design. These contracts enable a set of safety obligations to then be identified for each object in the system. By verifying that these obligations are met it is possible to demonstrate that the software system is acceptably safe.[4] The case study has shown how a coherent safety argument can be developed using the analysis techniques, resulting safety obligations and verification activities.

[4]The nature of the verification activities is outside the scope of this case study.

Chapter 7

Evaluation

7.1 Introduction

In chapter 2.1, the following proposition was stated:

> Through the development of safety contracts, it is possible to establish a systematic, thorough and scalable process to identify the properties required of software objects to adequately address their contribution to system-level hazards.

In chapter 2 it was stated that the thesis would support the proposition through:

1. The development of a hazard-driven, product based safety analysis process for OO software.
2. The integration of safety requirement elicitation as part of the process.
3. The development of safety argument patterns which supports the use of this approach.

This thesis has described the development of these three elements in much detail. In order to evaluate the extent to which this thesis addresses the proposition, it is necessary to assess how well the approach meets the criteria stated in the proposition. These are that the approach should be:

- Systematic
- Thorough
- Scalable

The thesis will also be evaluated against the problem statements presented in section 2.4.

7.2 Systematic Approach

A systematic approach can be described as one which is "Arranged or conducted according to a system, plan, or organised method" [72]. This is crucial to the effectiveness of the approach proposed in this thesis, as the intention is that the approach may be adopted by engineers wishing to utilise an OO approach for a safety critical application. One of the weaknesses identified in the existing literature relating to the area was seen to be the lack of such a systematic method.

A number of techniques were identified for analysing aspects of OO designs, as well as others which could be adapted for use in such analysis. Although providing useful output in isolation, no guidance could be found in the literature on when the use a particular technique was required, or indeed what the purpose or intended output of each technique was. As well as making it difficult for a system analyst to identify which techniques might be appropriate, crucially, it is also very difficult from existing literature to know which analysis is *necessary* in order to ensure that complete information is used in deriving safety requirements. This of course will therefore affect the level of rigour which is achieved.

Given this weakness in current literature, one aim of this thesis is to provide a coherent step-by-step process which can be followed by safety engineers when analysing OO designs. For each step of the process it should be clear which techniques are being used, which elements of the design are being analysed, and the outcome which is generated. This thesis uses the concept of safety contracts in order to provide the necessary structure. The analysis is focussed upon safety contract construction, giving each aspect of the analysis a clear and defined purpose. Each step of the analysis should provide information which is necessary in the construction of safety contracts, any analysis which does not provide such information, although perhaps useful for other reasons, is not considered necessary as part of the safety process. The safety contracts also form the basis of the safety arguments which are produced. The approach proposed in this thesis, through the exploitation of safety contracts, therefore provides the required integration between the different parts of the process.

In this thesis, chapter 3 identified the steps of the process, and the techniques best suited to that analysis. A small worked example of an SMS has been used to illustrate clearly how the analysis at each step should be performed. In identifying the techniques required, the purpose

of each analysis step has first been determined, and the information which must be captured as a result of the analysis is made clear. This ensures that all the information required for the construction of safety requirements is taken forward to the requirements definition stage.

For example, section 3.8 identifies the purpose of the analysis step as identifying the potential causes of interaction failures identified at the previous step. A SHARD-style analysis is identified as the technique to be used, and a suggested output format which captures all the required information is presented in figure 3.9. The SMS example is used to illustrate how the analysis is performed. Each step of the process defined in chapter 3 follows this format, such that it is possible for the reader to systematically work through each step.

The fact that each step in the process is clearly defined also brings other advantages. For example it facilitates a review of the process, which could be undertaken by using a checklist-based approach to identify which aspects of the analysis require further work. The fact that the requirements of each analysis step are clearly defined also makes it easier to provide automation for those parts of the analysis where such assistance may be found to be useful.

To illustrate how the approach may be applied in practice, the process is applied to a more realistic case study in chapter 6. The case study is helpful for analysts who wish to reproduce the analysis process for their own system design, as it demonstrates how the entire process can be applied and how the resulting evidence is used in supporting a safety argument which can be used as part of the certification of the system.

One of the key characteristics of the approach presented in this thesis is that each stage of the analysis process is performed for a specific reason, that is to generate safety contracts which can be used to provide evidence which is required to support the safety claims made about the system. Chapter 5 provides safety argument patterns which can be instantiated for any OO system to which the approach has been applied. This makes it straightforward, as illustrated with the case study in section 6.5, to produce a compelling argument as to the safety of the OO system.

Based on the evaluation above, it is possible to conclude that the approach developed in this thesis is systematic in nature, in that it presents an organised, structured and coherent method which could be reproduced successfully by others.

7.3 Thorough Approach

A thorough approach can be described as one which is “applied to or affecting every part or detail” [72]. This implies a level of completeness of the process, which in this case means that the process identifies and mitigate all possible contributions of the software to the system level hazards. Although it is extremely difficult to prove completeness, the effectiveness of the approach can be assessed by the level of thoroughness which is achieved. In order to be able to make a claim about the safety of a system, the potential contribution of all aspects of the system’s behaviour must be considered, and mitigated where necessary.

Considering the process described in this thesis, a hazard-driven approach is proposed. The starting point for the analysis is the hazards which have been identified for the system, then the causes of the hazards are identified. The advantage of such an approach is that it ensures that the analysis which is performed is focussed upon those situations which will could lead to an accident. Whereas the number of possible accidents and the number of potential causes of the accidents may be very large, the number of hazards is generally a much smaller number. By addressing the causes of the hazards it is therefore much easier to ensure that all potentially unsafe behaviour is addressed.

One thing to note about the contribution of this thesis is that no new method is proposed for identifying system level hazards. Since the thesis is primarily concerned with the contribution of software to the system level hazards, the thesis does not concern itself with how such hazards are identified. Since the systems in which OO software may be used are likely to be either existing systems, or systems which are well understood, the hazards which are present for the system are also likely to be well understood. There is therefore an explicit assumption that the list of system hazards, which is used as the starting point of the process is complete and accurate. This assumption is captured in the safety argument, as shown in the software system level argument in Appendix A. Consequently if the hazard list were incomplete then this assumption would be broken and the safety argument would not be valid.

Given that all the hazards associated with the system have been correctly identified, the contribution that may be made by the software to each hazard must then be identified. This is done by considering all types of failure of the software. The analysis in chapter 3 considers three separate groups of failures, these are the function, timing and value. These failure groups are based on a classification used by Pumfrey, based on the models of Bondavalli and Simoncini [6], and Ezilchelvan and Shrivastava [16]. These failure classifications are obtained by considering the behaviour of the software in terms of services which it provides. As such, this classification

can be considered to provide coverage of all the potential failure behaviour of the software.

For each failure type the behaviour which may contribute to each hazard is identified using a combination of techniques, this includes the use of both *deductive* analyses such as FTA, and *inductive* analyses such as SHARD. It should be noted that the analysis techniques proposed in this thesis analyse different views of the software design. This identifies a limitation of the approach which is that the completeness of the analysis is heavily dependent upon the expressiveness and representativeness of the view used for the analysis. If interactions are not included in the dynamic view of the system for example, then their potential impact on hazardous behaviour could not be fully analysed. It was identified in section 3.5.4.2 that this is also an important issue when considering the completeness of statechart analysis. It is perhaps also worth noting at this point that providing a rigorous evaluation of the thoroughness of a safety analysis methodology is very difficult. Even if the approach proposed in this thesis were applied to a real system which is then commissioned, it would only be through the analysis of accidents and incidents which occur that any feedback on the rigour of the approach could be obtained. Given the exceptionally low accident rates achieved by modern safety-critical systems, it would be impossible to obtain results which are statistically significant.

Despite this, the approach presented in this thesis is based on sound principles which guarantee a level of thoroughness necessary to form a convincing safety argument.

7.4 Scalability

A key consideration in the effectiveness of the approach is its ability to be taken and applied to large scale software systems. Although a case study was used in chapter 6 to demonstrate the feasibility of application to a larger example system, the question still remains as to how well the process can be applied as the size and complexity of the software system increases.

A key thing to note in answer to this problem is that the effort required to carry out the process is as much linked to the number of hazards that have been identified for a particular system, as it is to the size of the software. It is, for example, quite possible that a fairly small software design used in the context of a system with a large number of hazards would require similar effort to perform the analysis as a large and complicated piece of software being used within a system with only a few hazards.

It should also be noted that since the process is hazard-driven, it is only those parts of the design which are identified as potentially contributing to a hazard which require extensive

investigation. The effect of this is that just because a software system is very large, does not imply that a large amount of analysis is necessarily required. In many cases large parts of the software may not be involved in any safety critical activities. The process developed in this thesis enables the analysis to be focussed upon only those areas which may contribute to hazardous behaviour. The amount of effort required in the analysis will therefore increase in proportion to the number of hazards present. The process defined, however, allows each hazard to be treated separately, and therefore it is possible to address a system with a large number of hazards through a proportional increase in the size of the analysis team. The amount of analysis effort required for each hazard is then dependant upon the number of potential failures identified in the design relating to that hazard. Although this *is* likely to increase with the complexity of the design, again it is only those aspects of the design relating to that particular hazard which need to be considered, and not the entire system.

The discussion above explains how complex software does not *necessarily* lead to a huge increase in the amount of effort required to follow the thesis approach. However, should large amounts of analysis be required it is contended that this remains feasible. An increase in design complexity may lead to an increase in the amount of analysis required, however the complexity of that analysis should not increase. For example, the number of interactions requiring investigation may become large, but the method used for each interaction does not become more complex. The systematic nature of the approach enables large scale analysis to be conducted in a thorough way.

The one aspect of the process where the complexity of the design will affect the complexity of the analysis performed is the state chart analysis. It can be seen through the very simple examples provided in the thesis that this analysis can start to become intractable quite quickly. It is for this reason that tool support has been used to automate this aspect of the analysis. Using the toolset discussed in section 3.5.4.3 it is possible to perform the described analysis of more complex statecharts.

Another property of the proposed approach which aids its scalability arises from the ability to be able to use safety contracts to identify the safety obligations on individual classes of objects, as described in chapter 4. What this means is that the verification effort required to demonstrate the safety obligations are met by the software can be easily split up amongst a number of people. Since the obligations are so clearly broken down, it is possible for the evidence for one part of the design to be generated independently of other parts. In addition, the modular structure of the safety argument allows the argument for each class to be developed as a separate module,

this again facilities the partitioning of effort when developing large systems.

Producing figures (even realistic estimates) for the effort required to apply the process described in this thesis to a software system is impossible. In any case no corresponding figures for other approaches are available which can be used in estimating the relative feasibility of the approach for large systems. The discussion above described features of the approach which enable the author to have sufficient confidence that the approach could be applied to software systems of the complexity that may be encountered in safety-critical systems.

7.5 Evaluation Against Problem Statements

In chapter 2, based on the survey of relevant literature, a number of problems were identified which the thesis should address. In this section the approach developed in this thesis is evaluated against each of these problem statements.

Problem Statement 1 Ensuring the safety of an OO software system requires that the contribution of the software to system level hazards be identified and mitigated.

This thesis has developed a hazard-driven approach which identifies hazardous failure modes for the software system. The thesis has shown how the ways in which the software might fail, and lead to these failure modes, can be identified, and how such failures may be prevented through the use of safety contracts.

Problem Statement 2 There exist a number of techniques for analysing OO software designs, however, there is a lack of a coherent process.

This thesis has presented a process developed based on a number of analysis requirements which are necessary to meet one overall aim, that of identifying behaviour of the software that may contribute to a system hazard. The techniques used are identified based on their ability to generate particular desired results as part of achieving this aim. In this way this thesis demonstrates how the techniques may be used in combination in order to ensure the safety of the developed system.

Problem Statement 3 As part of any such process, it is necessary that safety requirements may be generated in a way that supports the OO paradigm.

It would be possible to specify the DSRs arising from the analysis in any number of ways. Chapter 4 identified contracts as an ideal way of capturing requirements for OO systems, since

they provide excellent support for key features of OO systems such as inheritance. The analysis process described in this thesis has been specifically developed to facilitate the representation of DSRs in the form of safety contracts between objects.

Problem Statement 4 There exists little guidance on how a defensible safety argument may be produced for an OO system based on the use of a combination of techniques.

This thesis has demonstrated how it is possible to structure a defensible safety argument for an OO software system. This does not mean that the method proposed in this thesis provides a proof of the safety of the resulting system, nor was it the intention of this thesis to do so. As with existing approaches to software safety, this method proposes an argument structure to be developed which argues only the acceptability of the software. Acceptability is a judgement which is made based upon the rigour of the evidence provided about the safety of the system, and the level of risk associated with the system hazards. The safety argument patterns which have been developed identify how the evidence generated throughout the process described in the thesis can be used to support the safety argument claims.

Problem Statement 5 It is essential that safety process can be successfully integrated with existing development processes.

In chapter 2, it was identified as being crucial to the success of a safety process that it is integrated with the development process, rather than being seen as a separate activity. In order to achieve this, the process must be able to be integrated with the existing development processes used for the software. The approach developed in this thesis has achieved this aim in two ways.

Firstly, through focusing the analysis on design artifacts, rather than on particular steps in the design process. This enables the safety process to fit within an existing development process rather than imposing any particular process. Secondly, the approach is not specifically dependent upon any particular design methodology. Although the example design artifacts used in the thesis are generally in UML (being by far the most common notation), the use of UML is not required, and the same approach is applicable whatever notation is used.

7.6 Conclusions

The thesis proposition stated that it was possible through the use of safety contracts to establish an effective process to identify the properties required of software objects to adequately address

their contribution to system-level hazards. In this chapter the extent to which this thesis addresses the proposition has been evaluated. Firstly, the approach was shown to satisfy the criteria for an effective process which were set out in the thesis proposition. It was also shown how the thesis addressed the problem statements identified from the literature survey. This suggests that the approach developed in this thesis is an effective one, however it is only through the extended practical application of the approach to real projects that a full evaluation of the approach's effectiveness can be made.

Chapter 8

Conclusions

8.1 Concluding Remarks

This thesis has developed a systematic, thorough and scalable process to identify the properties required of software objects to adequately address their contribution to system-level hazards. Specifically, the contribution of the work presented in this thesis lies in the following three areas:

- Definition of a coherent hazard-driven process for the rigorous analysis of OO software designs.
- Use of the safety contract concept to elicit safety requirements in a manner which supports OO features.
- Development of safety argument patterns for making defensible claims about the safety of the resulting software system.

In the remaining sections, some conclusions are drawn from each of these areas of research, and finally in section 8.2 some areas worthy of further work are proposed.

8.1.1 Conclusions on the analysis process contribution

The key contribution made by the safety analysis process described in chapter 3 is to provide a structured approach to the analysis of OO software designs. The process largely makes use of existing analysis techniques, however crucially provides a framework in which each technique makes a specific and clear contribution to the overall objectives of the process. As well as identifying potential contributions to hazardous failure modes, these objectives also include the

derivation of safety requirements in safety contract form. Through focussing on interactions between objects, the analysis process facilitates the development of safety contracts.

The analysis process has been argued to be systematic, thorough and scalable in chapter 7, and its applicability to an industrially sized design has been demonstrated in the case study presented in chapter 6.

8.1.2 Conclusions on the use of safety contracts

The use of safety contracts provides an important contribution in a number of ways. Firstly the use of contracts has been shown to help support maintainability, inheritance and reuse within OO designs. The use of safety contracts therefore helps to ensure that the safety activities do not overly detract from the potential benefits of using an OO approach.

Just as importantly, however, safety contracts provide a link between the analysis process and the safety argument. The safety argument structure proposed in chapter 5 is structured in a modular fashion, with separate modules of argument for the interactions which occur between classes, and each of the classes themselves. This structure brings many advantages, which were discussed in detail in chapter 5. It is the use of safety contracts which allow such a modular argument structure to be used, by enabling the claims to be modularised in this manner. Through developing the safety contracts, the analysis process therefore identifies the evidence which is required by the safety argument.

It has been shown how the properties required for safety contracts can be represented using OCL notation, which provides a formal and side-effect free expression language. Using OCL the safety contracts can be integrated as part of the design, facilitating their implementation.

8.1.3 Conclusions on the safety argument patterns

The contribution provided by the development of the safety argument patterns is that they show explicitly how a defensible argument for an OO software system may be structured. No such guidance exists elsewhere. The case study in chapter 6 illustrated how a safety argument can be constructed through appealing to specific evidence generated through following the process described in this thesis.

As discussed previously, the modular structure of the argument has been chosen such that it may minimise the effect of changes to the design, and maximises the amount of argument that may be reused.

8.2 Further work areas

Whilst carrying out the research for this thesis a number of areas were identified which are worthy of further investigation. The following sections briefly introduce some of these areas of research.

8.2.1 Verification Evidence

One important aspect of demonstrating the safety of an OO software system which has not been investigated as part of this thesis is verification methods. The literature survey in chapter 2 explored some of the available verification techniques in order to ensure the process was amendable to verification. It was however outside of the scope of this thesis to determine the most effective verification approach. One beneficial area of future work would therefore be to investigate different verification strategies for OO software to determine which strategy was the most effective at providing the evidence required to support the safety argument. This work could include the development of an argument pattern relating to verification which could be incorporated as part of the Class Argument Module Pattern.

8.2.2 Safety Contract Enforcement

It is ultimately through demonstrating that the obligations in the safety contracts are met by the implemented software that the software system is shown to be acceptably safe. One interesting area of future work would be to investigate ways in which the defined safety contracts could be enforced. This could potentially be achieved, for example, through run-time checks made by the software that the contractual obligations are met each time an operation call is made. Another approach may be to enforce the contracts during implementation. This would be particularly useful in situations where code was being automatically generated from the design. By integrating the contractual obligations into the design, it may then be possible to guarantee that these constraints are met by the implementation. There are many potential problems with both of these approaches, however they would be interesting avenues of further research.

8.2.3 System Implementation

This thesis has described a process for developing safe OO systems. A case study has been used to demonstrate how this process can be used to derive safety requirements in the form of safety

contracts upon the design of the system, and how these contracts can then be used in creating a safety argument for the resulting software. One area that was not considered as part of this case study was the implementation of the developed software design. Implementation was explicitly outside of the scope of the thesis, however there are many very interesting issues (particularly relating to object-oriented programming languages) which are worthy of further investigation. There is much existing research in the area of OO programming, however there is almost certainly scope for investigating further the challenges associated with the implementation of designs containing safety contracts.

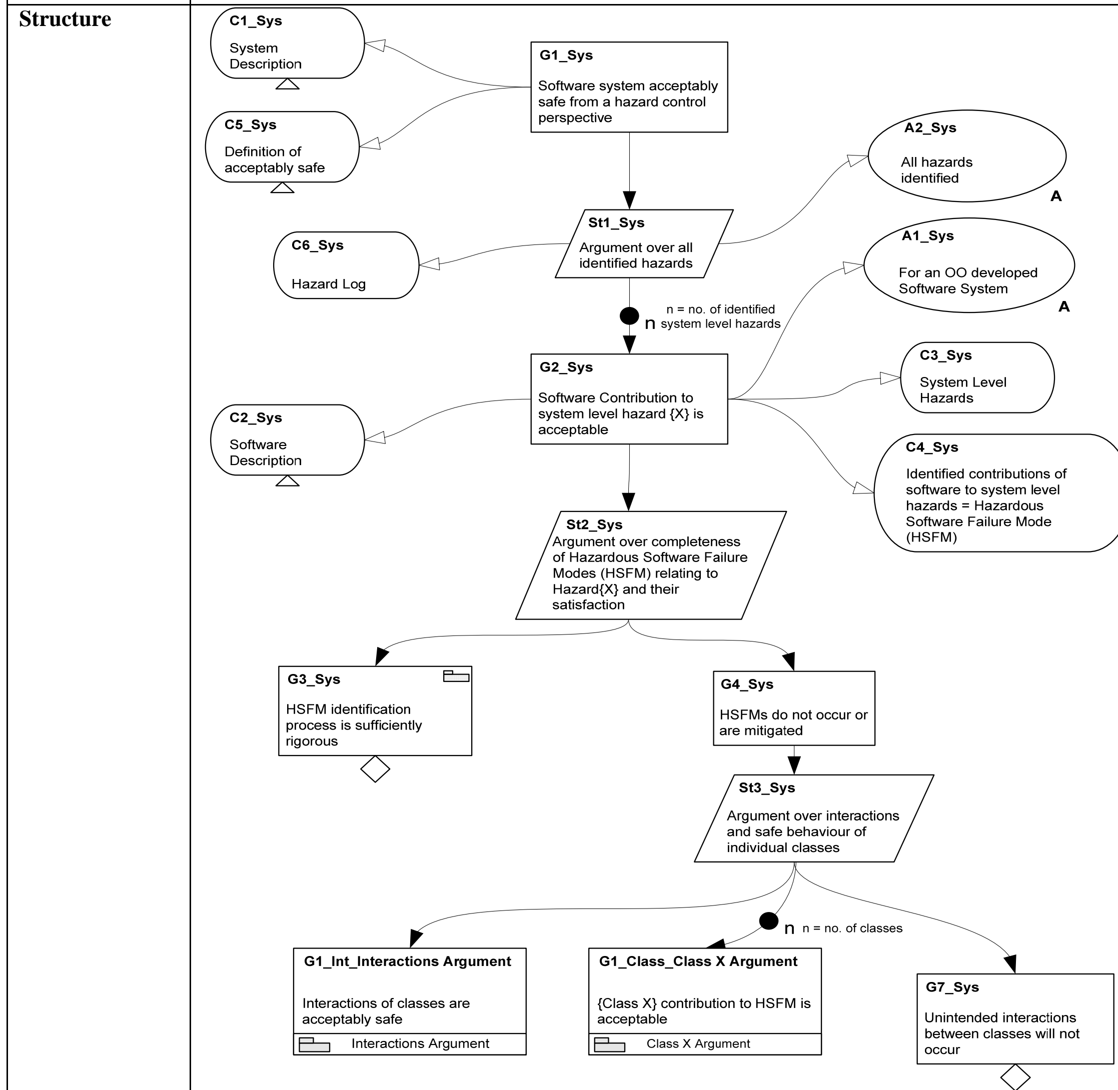
8.3 Overall conclusions

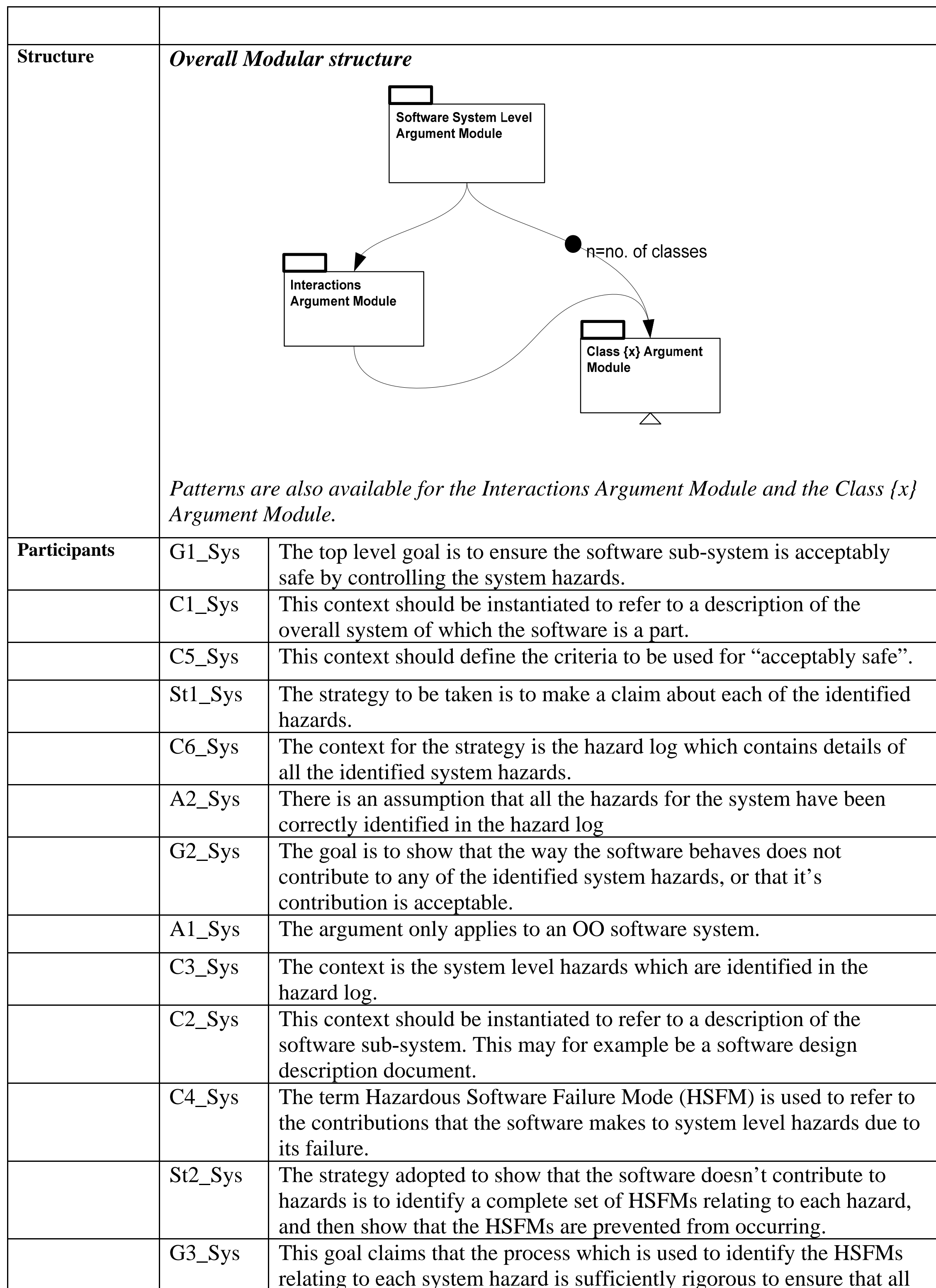
This thesis did not set out to champion the use of the OO paradigm for safety critical systems, or to suggest that its use could in any way lead to safer systems. The starting point was very much with an assumption that the desire to use an OO approach already existed. Given this, the thesis set out to identify what problems presented themselves when attempting to use OO for safety critical systems. A number of problems were identified which this thesis has addressed. In doing so, the author hopes that the work presented in this thesis may enable those otherwise deterred from pursuing an OO approach when developing safety critical systems to do so.

Appendix A

Software System Argument Module Pattern

Software System Level Argument Module			
Author	Richard Hawkins		
Created	21/03/06	**Last Modified**	21/03/06

Intent	The intent of this pattern is to create top level arguments for an OO software system
Also Known As	
Motivation	It has been shown that a modular argument structure is advantageous for OO software. This argument module provides the top level argument for the software system as a whole, which appeals to the arguments made in the modules for the interactions and the individual classes.
Structure	

Structure	***Overall Modular structure*** 	

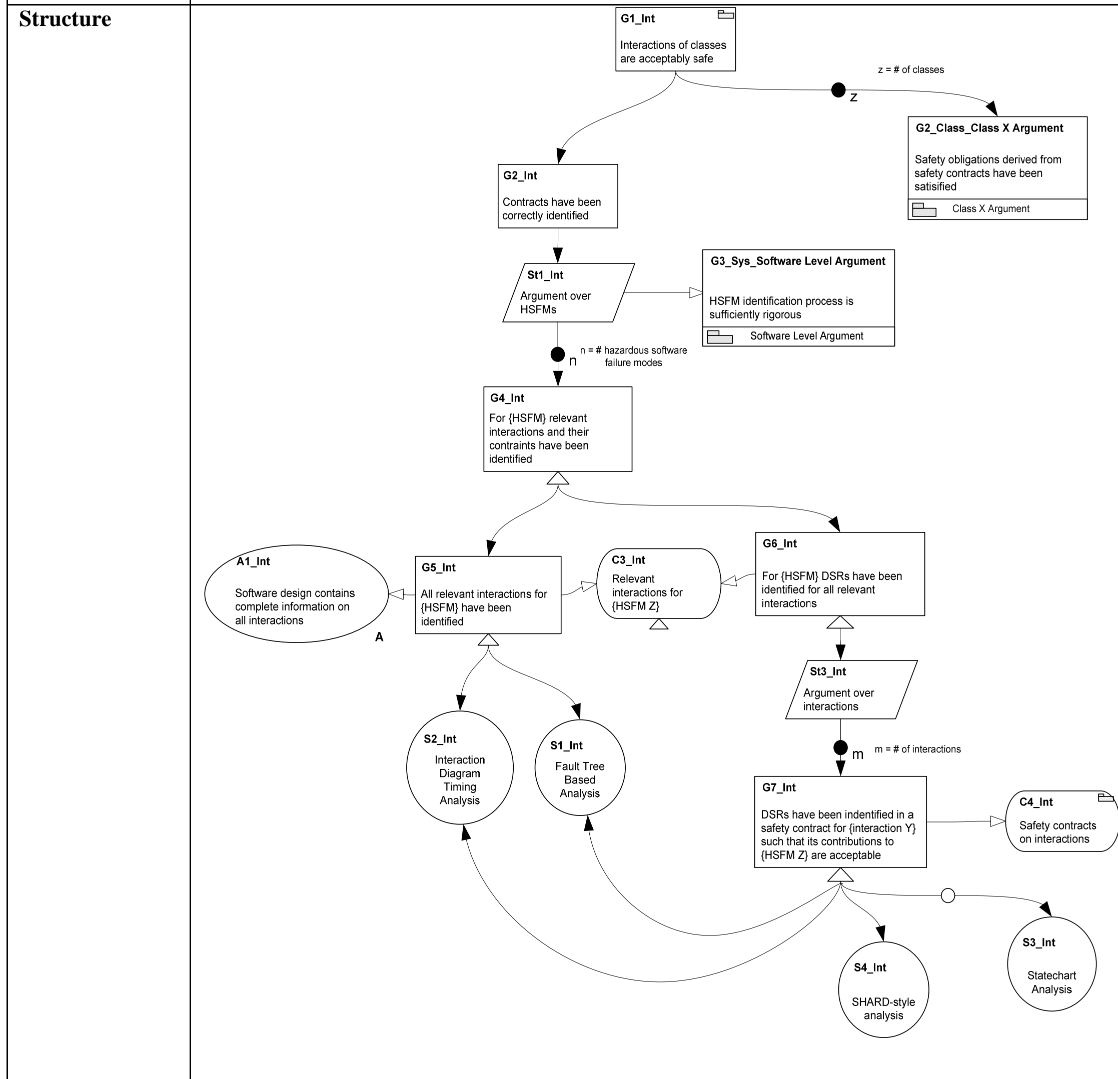
	Patterns are also available for the Interactions Argument Module and the Class {x} Argument Module.	
Participants	G1_Sys	The top level goal is to ensure the software sub-system is acceptably safe by controlling the system hazards.
	C1_Sys	This context should be instantiated to refer to a description of the overall system of which the software is a part.
	C5_Sys	This context should define the criteria to be used for “acceptably safe”.
	St1_Sys	The strategy to be taken is to make a claim about each of the identified hazards.
	C6_Sys	The context for the strategy is the hazard log which contains details of all the identified system hazards.
	A2_Sys	There is an assumption that all the hazards for the system have been correctly identified in the hazard log
	G2_Sys	The goal is to show that the way the software behaves does not contribute to any of the identified system hazards, or that it’s contribution is acceptable.
	A1_Sys	The argument only applies to an OO software system.
	C3_Sys	The context is the system level hazards which are identified in the hazard log.
	C2_Sys	This context should be instantiated to refer to a description of the software sub-system. This may for example be a software design description document.
	C4_Sys	The term Hazardous Software Failure Mode (HSFM) is used to refer to the contributions that the software makes to system level hazards due to its failure.
	St2_Sys	The strategy adopted to show that the software doesn’t contribute to hazards is to identify a complete set of HSFMs relating to each hazard, and then show that the HSFMs are prevented from occurring.
	G3_Sys	This goal claims that the process which is used to identify the HSFMs relating to each system hazard is sufficiently rigorous to ensure that all

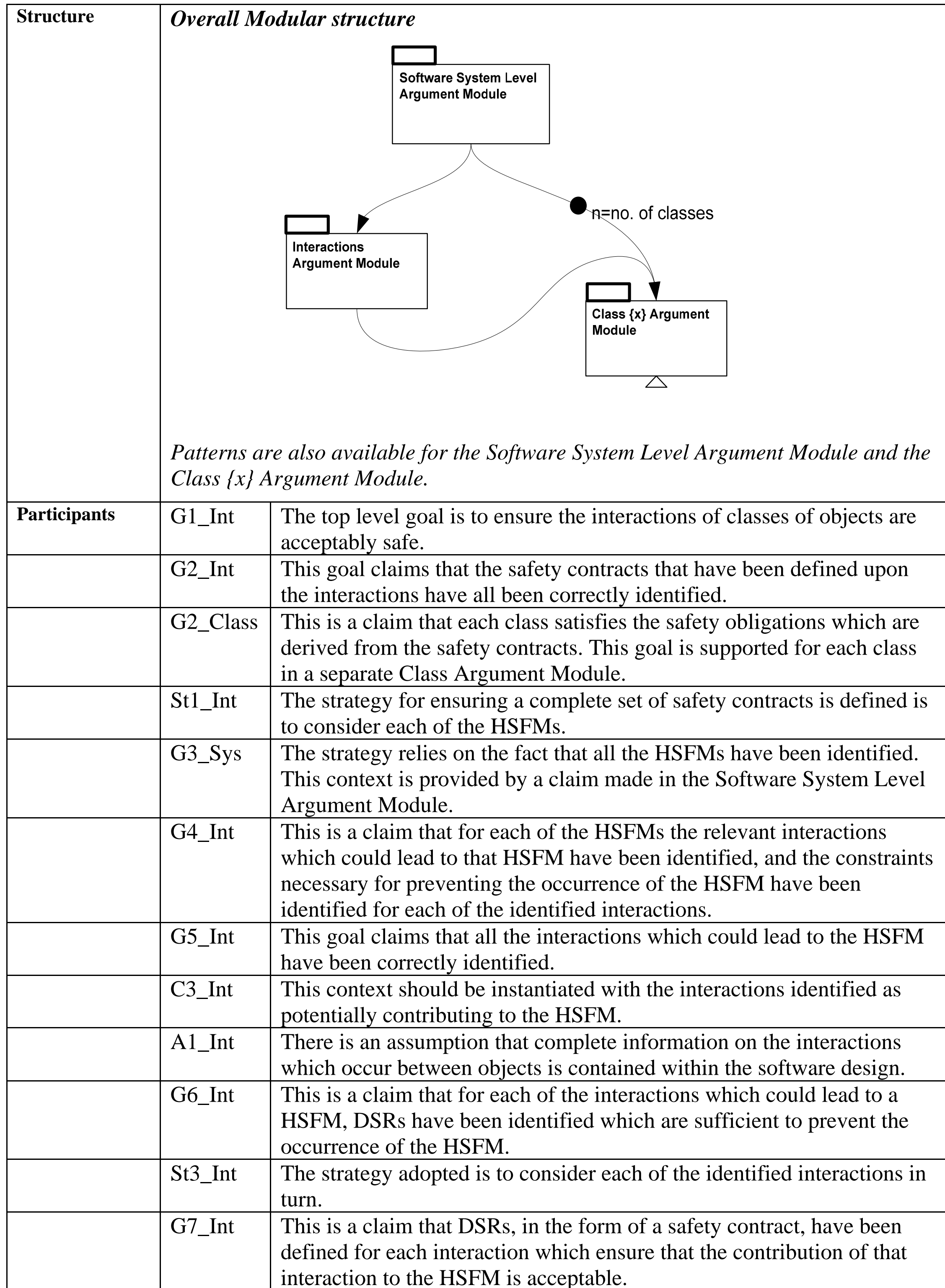
		HSFMs are correctly identified.
	G4_Sys	This claims that none of the HSFMs which have been identified for the software cannot occur, or if they can, their effects are suitably mitigated.
	St3_Sys	In order to show that the HSFMs do not occur, the strategy adopted is to argue about the safety the safety of the behaviour of the objects of each of the classes in the software design, and of the interactions which occur between those objects. It is also necessary to show that there are no unintended interactions between the objects.
	G7_Sys	This goal claims that unintended interactions between objects will not occur.
	G1_Int	This is a claim about the safety of the interactions between the objects and forms part of the Interactions Argument Module.
	G1_Class	This is a claim about the contribution of the individual classes to the HSFMs. One such goal is required for each of the classes in the software design. Each of the goals forms part of a separate Class Argument Module.
Collaborations	This pattern provides one module of argument in a modular safety argument. The argument requires support from a number of other argument modules (Interactions Argument Module and one Class Argument Module for each class in the software design).	
Applicability	This pattern is only applicable to OO software systems which have been developed using a safety contract approach. Such an approach is required to make the necessary claims about the objects and their interactions.	
Consequences	After instantiating the pattern , a number of unresolved goals will remain: • **G3_Sys –** To support this claim evidence about the process which is used to identify the HSFMs must be presented. • **G7_Sys –** Verification evidence must be provided to support the claim that only interactions considered during the analysis can occur between the objects in the system. This evidence could for example be generated through testing.	
Implementation	***Possible Pitfalls*** • Attempting to apply the pattern to a software system which is not OO. • Attempting to apply the pattern without sufficient design information (static class structure) available. • Having incomplete or out of date details on system hazards or HSFMs.	
Examples	See *Software system level argument for the ACS case study*.	
Known Uses	See example above	
Related Patterns	Interactions Argument Module Pattern and the Class Argument Module Pattern both provide support for this pattern.	

Appendix B

Interactions Argument Module Pattern

Interactions Argument Module			
Author	Richard Hawkins		
Created	21/03/06	**Last Modified**	21/03/06

Intent	The intent of this pattern is to create an argument regarding the safety of the interactions between objects in an OO software system.
Also Known As	
Motivation	It has been shown that a modular argument structure is advantageous for OO software. This argument module provides the argument that the interactions are acceptably safe.
Structure	

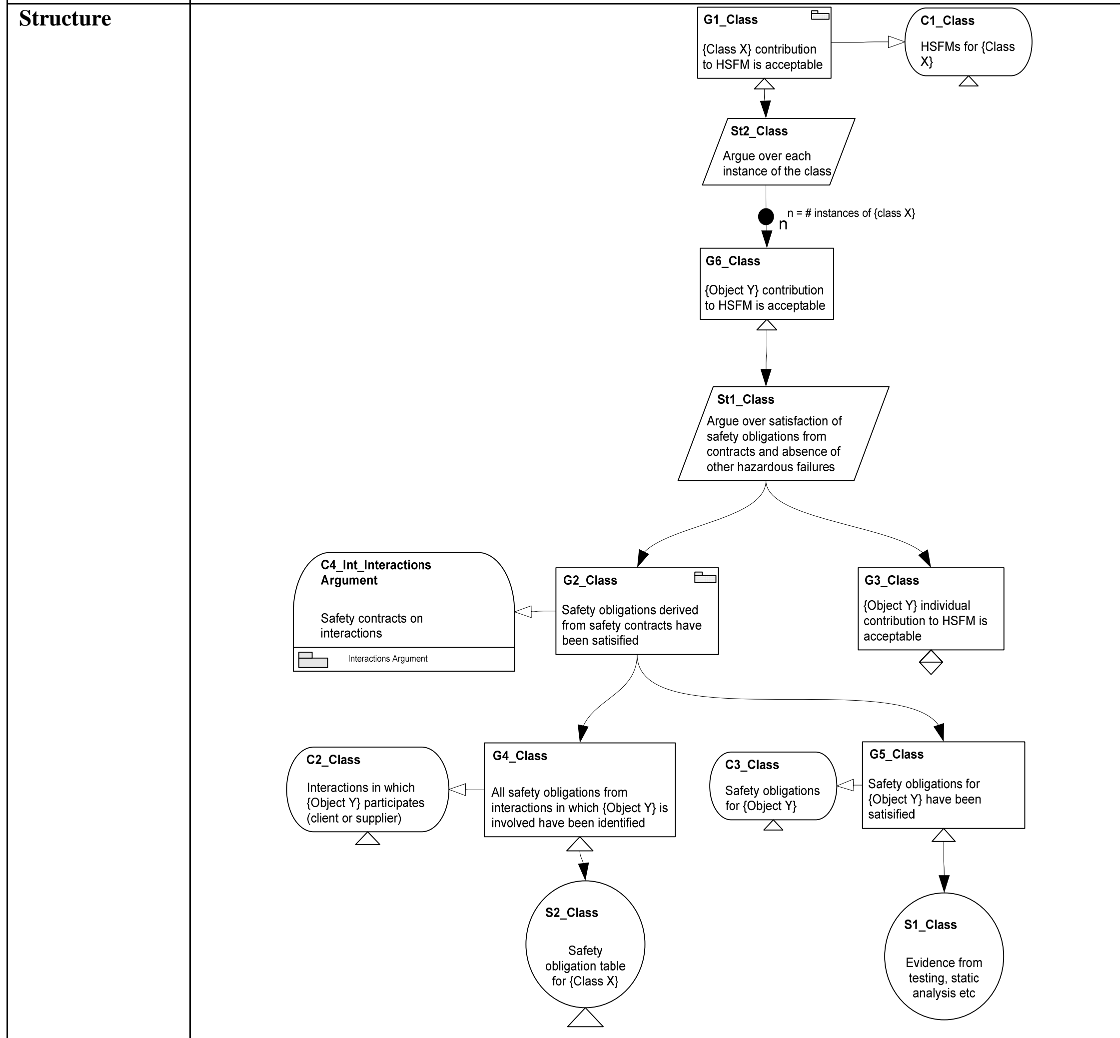
Structure	***Overall Modular structure***  *Patterns are also available for the Software System Level Argument Module and the Class {x} Argument Module.*	
Participants	G1_Int	The top level goal is to ensure the interactions of classes of objects are acceptably safe.
	G2_Int	This goal claims that the safety contracts that have been defined upon the interactions have all been correctly identified.
	G2_Class	This is a claim that each class satisfies the safety obligations which are derived from the safety contracts. This goal is supported for each class in a separate Class Argument Module.
	St1_Int	The strategy for ensuring a complete set of safety contracts is defined is to consider each of the HSFMs.
	G3_Sys	The strategy relies on the fact that all the HSFMs have been identified. This context is provided by a claim made in the Software System Level Argument Module.
	G4_Int	This is a claim that for each of the HSFMs the relevant interactions which could lead to that HSFM have been identified, and the constraints necessary for preventing the occurrence of the HSFM have been identified for each of the identified interactions.
	G5_Int	This goal claims that all the interactions which could lead to the HSFM have been correctly identified.
	C3_Int	This context should be instantiated with the interactions identified as potentially contributing to the HSFM.
	A1_Int	There is an assumption that complete information on the interactions which occur between objects is contained within the software design.
	G6_Int	This is a claim that for each of the interactions which could lead to a HSFM, DSRs have been identified which are sufficient to prevent the occurrence of the HSFM.
	St3_Int	The strategy adopted is to consider each of the identified interactions in turn.
	G7_Int	This is a claim that DSRs, in the form of a safety contract, have been defined for each interaction which ensure that the contribution of that interaction to the HSFM is acceptable.

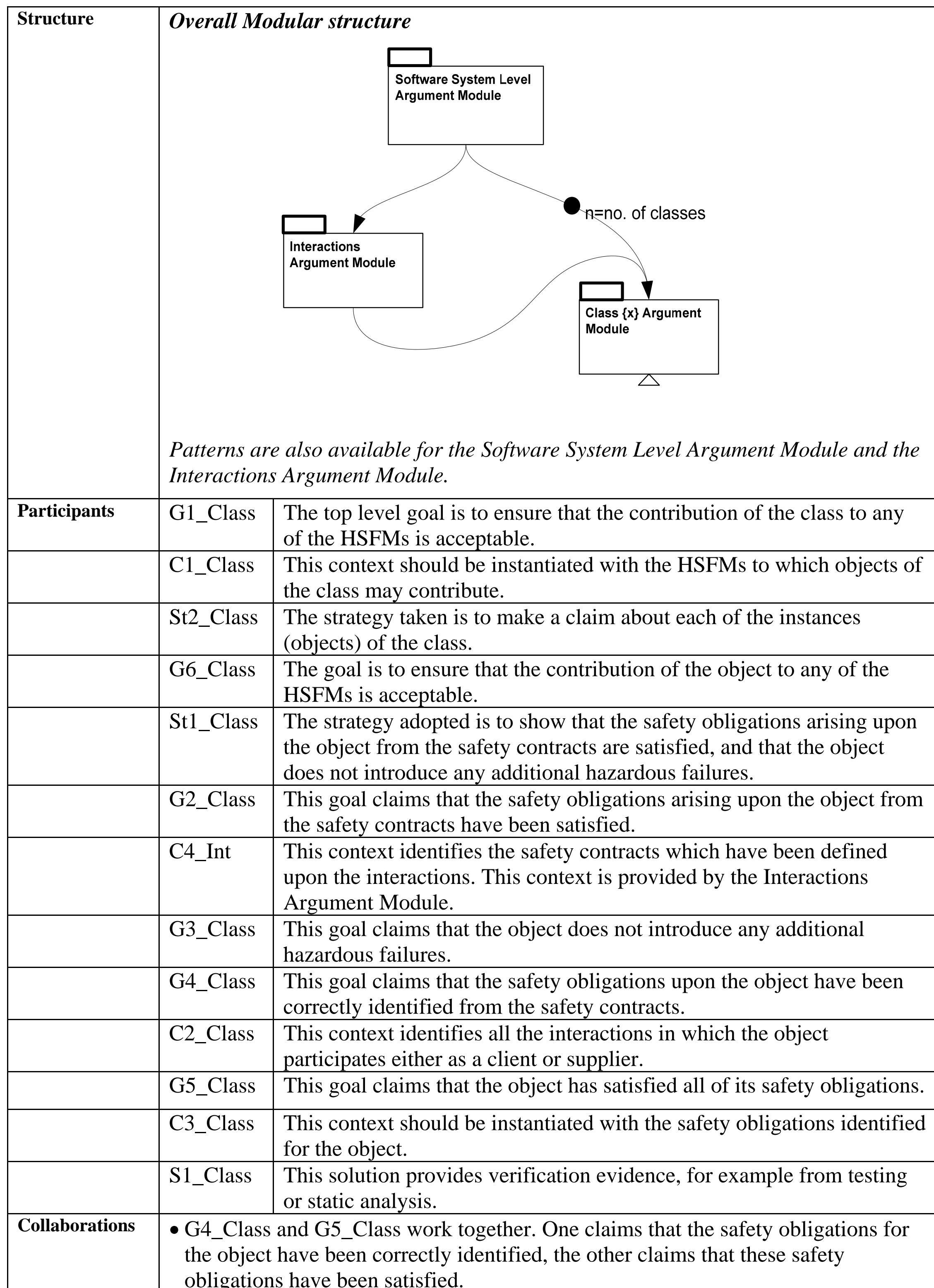
	C4_Int	This context should be instantiated with the safety contract defined for each interaction.
	S1_Int	This solution provides evidence gained from fault tree based analysis in support of goals G5_Int and G7_Int.
	S2_Int	This solution provides evidence gained from interactions diagram timing analysis in support of goals G5_Int and G7_Int.
	S3_Int	This solution can be used to provide evidence gained from statechart analysis in support of goals G7_Int. This solution is optional.
	S4_Int	This solution provides evidence gained from SHARD-stylet analysis in support of goals G7_Int.
Collaborations	• G2_Int and G2_Class work together. One claims that the contracts are correct, the other claims that the contracts are satisfied. • G5_Int and G6_Int work together. One claims that all the interactions which could lead to a HSFM have been correctly identified, the other claims that DSRs have been identified for each of those interactions in order to prevent the occurrence of the HSFM. • This pattern provides one module of argument in a modular safety argument. The argument requires support from a number of other argument modules (Software System Level Argument Module and one Class Argument Module for each class in the software design).	
Applicability	This pattern is only applicable to OO software systems which have been developed using a safety contract approach, and the analysis process described in this thesis.	
Consequences		
Implementation	***Possible Pitfalls*** • Attempting to apply the pattern to a software system which is not OO. • Attempting to apply the pattern without complete design information of the interactions (e.g. interaction diagrams) available. • Having incomplete or out of date details on system hazards or HSFMs. • Not using the analysis process described in this thesis.	
Examples	See *Interactions argument for the ACS case study*.	
Known Uses	See example above	
Related Patterns	Software System Level Argument Module Pattern and the Class Argument Module Pattern both provide support for this pattern.	

Appendix C

Class Argument Module Pattern

Class Argument Module			
Author	Richard Hawkins		
Created	21/03/06	**Last Modified**	21/03/06

Intent	The intent of this pattern is to create an argument regarding the safety of a class in an OO software system.
Also Known As	
Motivation	It has been shown that a modular argument structure is advantageous for OO software. This argument module provides the argument that a classes is acceptably safe. An argument should be created for each class in the software design for which safety obligations are identified.
Structure	

Structure	***Overall Modular structure***  *Patterns are also available for the Software System Level Argument Module and the Interactions Argument Module.*	
Participants	G1_Class	The top level goal is to ensure that the contribution of the class to any of the HSFMs is acceptable.
	C1_Class	This context should be instantiated with the HSFMs to which objects of the class may contribute.
	St2_Class	The strategy taken is to make a claim about each of the instances (objects) of the class.
	G6_Class	The goal is to ensure that the contribution of the object to any of the HSFMs is acceptable.
	St1_Class	The strategy adopted is to show that the safety obligations arising upon the object from the safety contracts are satisfied, and that the object does not introduce any additional hazardous failures.
	G2_Class	This goal claims that the safety obligations arising upon the object from the safety contracts have been satisfied.
	C4_Int	This context identifies the safety contracts which have been defined upon the interactions. This context is provided by the Interactions Argument Module.
	G3_Class	This goal claims that the object does not introduce any additional hazardous failures.
	G4_Class	This goal claims that the safety obligations upon the object have been correctly identified from the safety contracts.
	C2_Class	This context identifies all the interactions in which the object participates either as a client or supplier.
	G5_Class	This goal claims that the object has satisfied all of its safety obligations.
	C3_Class	This context should be instantiated with the safety obligations identified for the object.
	S1_Class	This solution provides verification evidence, for example from testing or static analysis.
Collaborations	• G4_Class and G5_Class work together. One claims that the safety obligations for the object have been correctly identified, the other claims that these safety obligations have been satisfied.	

	• This pattern provides one module of argument in a modular safety argument. The argument requires support from a number of other argument modules (Software System Level Argument Module and Interactions Argument Module).
Applicability	This pattern is only applicable to OO software systems which have been developed using a safety contract approach.
Consequences	After instantiating the pattern, the following unresolved goal remains: • **G3_Class –** To support this claim evidence must be provided about the absence of other failures such as subtle value failures.
Implementation	***Possible Pitfalls*** • Attempting to apply the pattern to a software system which is not OO. • Attempting to apply the pattern without having defined safety contracts upon interactions. • Having incomplete or out of date details on system hazards or HSFMs.
Examples	See *Navigator class argument for the ACS case study*.
Known Uses	See example above
Related Patterns	Software System Level Argument Module Pattern and the Interactions Argument Module Pattern both provide support for this pattern.

Appendix D

ACS Case Study Reference Material

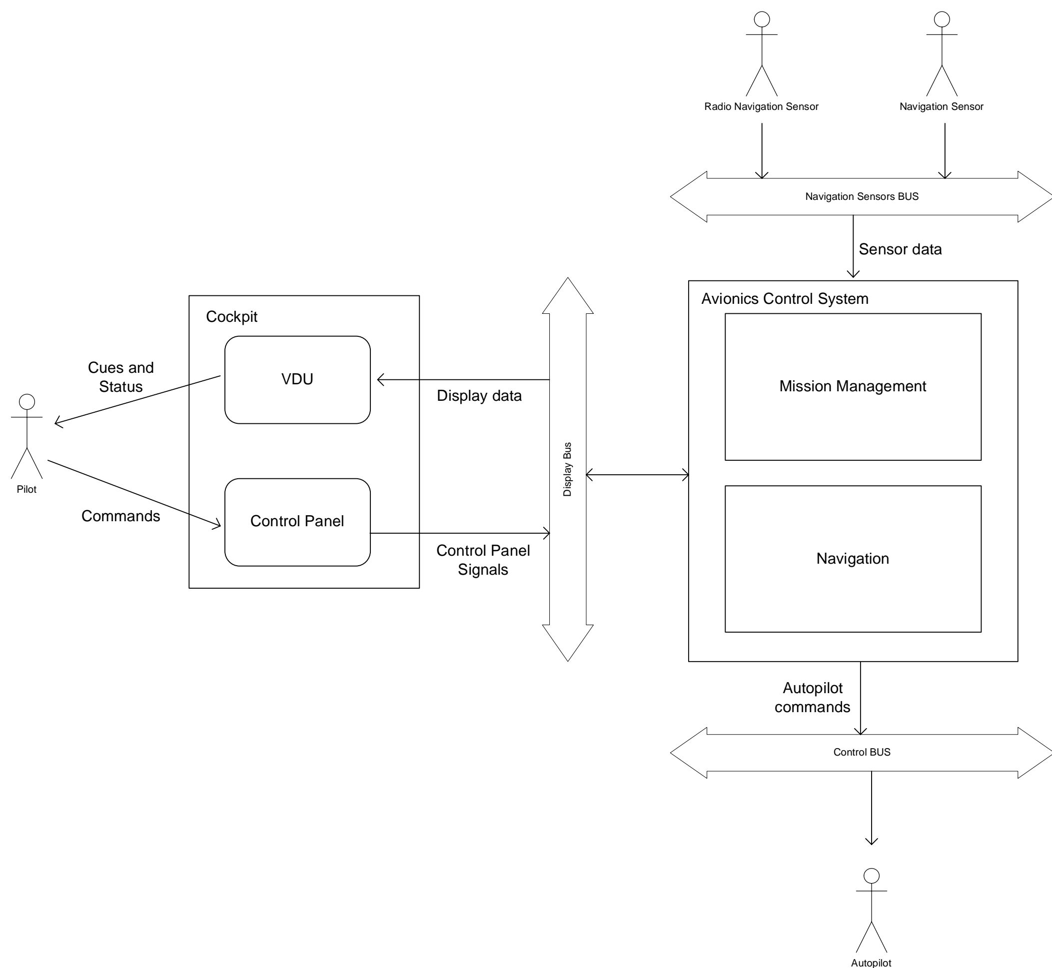
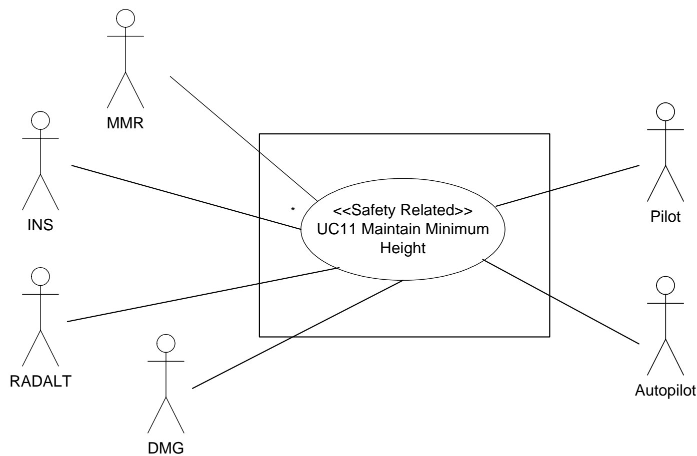


Figure D.1: System architecture diagram for the ACS



The pilot uses the control panel to set the minimum safe height and enable the TF mode. The system monitors the aircraft height (using the sensor configuration for TF navigation mode) and sends a TF radar cue to the pilot via the VDU. In addition the autopilot vertical command is updated and sent to the autopilot.

Pre Condition:
None

Flow:
•The pilot enables the TF mode via the control panel
•The pilot selects the minimum height via the control panel
•The minimum height is displayed on the VDU for the pilot
•The system monitors the navigation sensors in order to determine the aircraft height
•The system computes the pilot TF cue
•The TF radar cue is displayed on the VDU for the pilot
•The system computes the autopilot TF commands
•The autopilot vertical command is updated and sent to the autopilot

Figure D.2: Use case diagram: Maintain minimum height

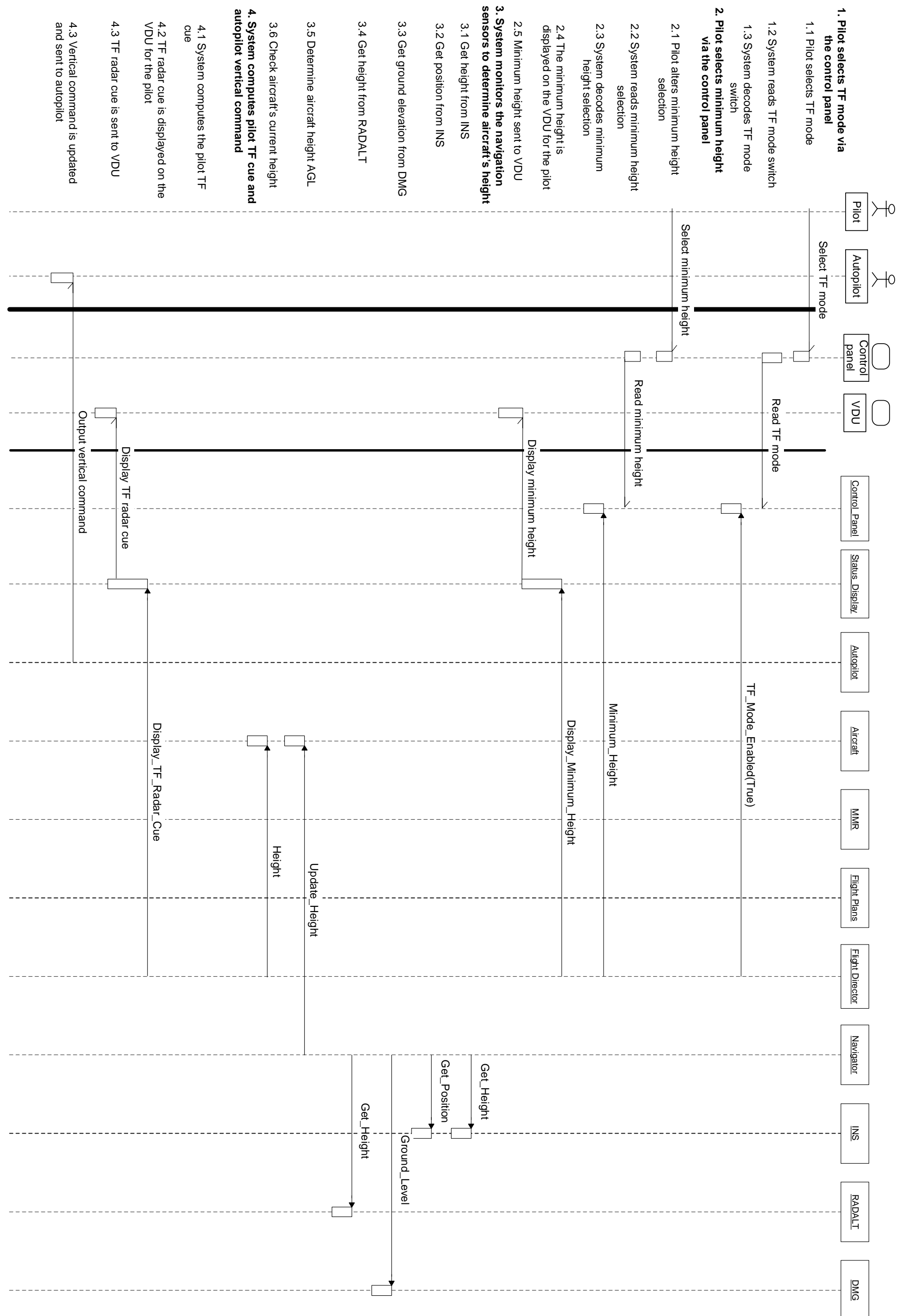


Figure D.3: Sequence diagram: Maintain minimum height - TF enabled

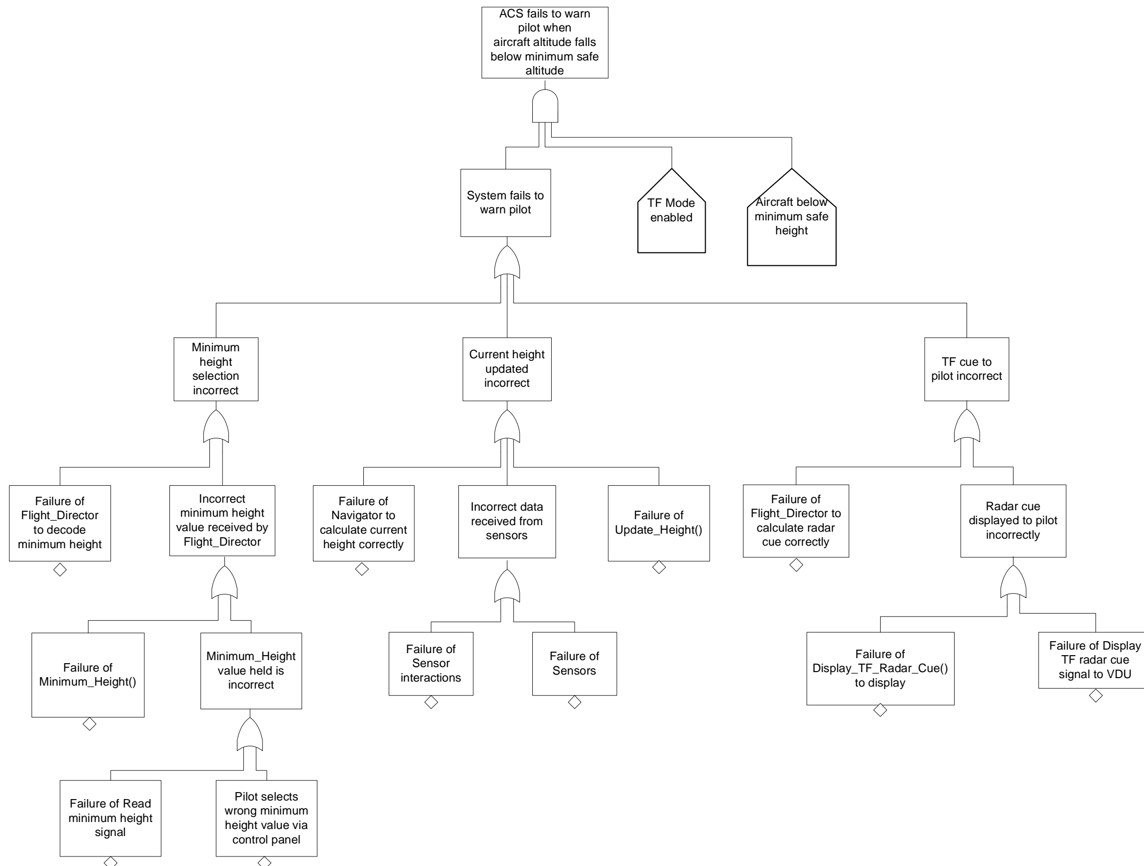


Figure D.4: Fault tree for ACS fails to warn pilot when aircraft altitude falls below minimum safe altitude

Interaction 1 – Min_Height () : Height_In_Feet

Client – *Flight Director*

Supplier – *Control Panel*

	Deviation	**Cause**	**Effect**	**Contribute to HSFM?**
OMISSION	1.1 Min_Height () call not made	Failure of client to send call	Height_In_Feet value not updated	Yes
	1.2 Height_In_Feet value not returned	Failure of supplier to return a value	Height_In_Feet value not updated	Yes
COMMISION	1.3 Min_Height () call made when not required	N/A	N/A	No
VALUE	1.4 Returned Height_In_Feet value is incorrect	Failure of supplier	Height_In_Feet value incorrect	Yes

Figure D.5: SHARD analysis of interaction 1

Interaction 2 – **Read minimum height signal**

Client – *Control Panel H/W*

Supplier – *Control Panel S/W*

	Deviation	**Cause**	**Effect**	**Contribute to HSFM?**
OMISSION	2.1 No signal is sent	Failure of communications, or Control panel H/W	Minimum height value not obtained	Yes
COMMISION	2.2 Spurious signal sent by client	Not known	Results in wrong Minimum height value (however must be *valid* spurious signal)	Yes
VALUE	2.3 Signal sent represents incorrect minimum height	Failure of Control panel H/W	Minimum height value incorrect	Yes

Figure D.6: SHARD analysis of interaction 2

Interaction 3 – Sensor Interactions : Get_Height() : Height_In_Feet – INS, RADALT
Get_Position() : Position – INS
Ground_Level() : Ground_Level - DMG

Client – *Navigator*

Supplier – *Listed above*

	Deviation	**Cause**	**Effect**	**Contribute to HSFM?**
OMISSION	3.1 Get_x () call not made	Failure of client to make call	Current value not updated	Yes
	3.2 Sensor value not returned	Failure of supplier to return a value	Current value not updated	Yes
COMMISION	3.3 Get_x () call made when not required	N/A	N/A	No
VALUE	3.4 Returned sensor value is incorrect	Failure of supplier	Sensor value incorrect	Yes

Figure D.7: SHARD analysis of interaction 3

Interaction 4 – Update_Height (in Height_In_Feet) : void

Client – *Navigator*

Supplier – *Aircraft*

	Deviation	**Cause**	**Effect**	**Contribute to HSFM?**
OMISSION	4.1 Update_Height () call not made	Failure of client to make call	Height not updated	Yes
	4.2 Height_In_Feet parameter missing	Failure of client to supply parameter	Unknown	Yes
COMMISION	4.3 Update_Height () call made when not required	N/A	N/A	No
VALUE	4.4 Parameter incorrect	Failure of client	Height value updated incorrectly	Yes

Figure D.8: SHARD analysis of interaction 4

Interaction 5 – Display_TF-Radar_Cue (in TF_Radar_Cue) : void

Client – *Flight director*

Supplier – *Status Display*

	Deviation	**Cause**	**Effect**	**Contribute to HSFM?**
OMISSION	5.1 Display_TF_Radar_Cue() call not made	Failure of client to make call	Radar cue not displayed	Yes
	5.2 TF_Radar_Cue parameter missing	Failure of client to provide parameter	Unknown	Yes
COMMISION	5.3 TF_Radar_Cue() call made when not required	N/A	N/A	No
VALUE	5.4 Parameter incorrect	Failure of client	Radar cue displayed incorrectly	Yes

Figure D.9: SHARD analysis of interaction 5

Interaction 6 – Display TF radar cue signal

Client – *Status Display*

Supplier – *VDU*

	Deviation	**Cause**	**Effect**	**Contribute to HSFM?**
OMISSION	6.1 No signal is sent	Failure of communications, or client S/W	Radar cue not displayed to pilot	Yes
COMMISION	6.2 Spurious signal sent by client	Not known	Results in incorrect radar cue being displayed (however must be *valid* spurious signal)	Yes
VALUE	6.3 Signal sent represents incorrect radar cue	Failure of client	Radar cue incorrect	Yes

Figure D.10: SHARD analysis of interaction 6

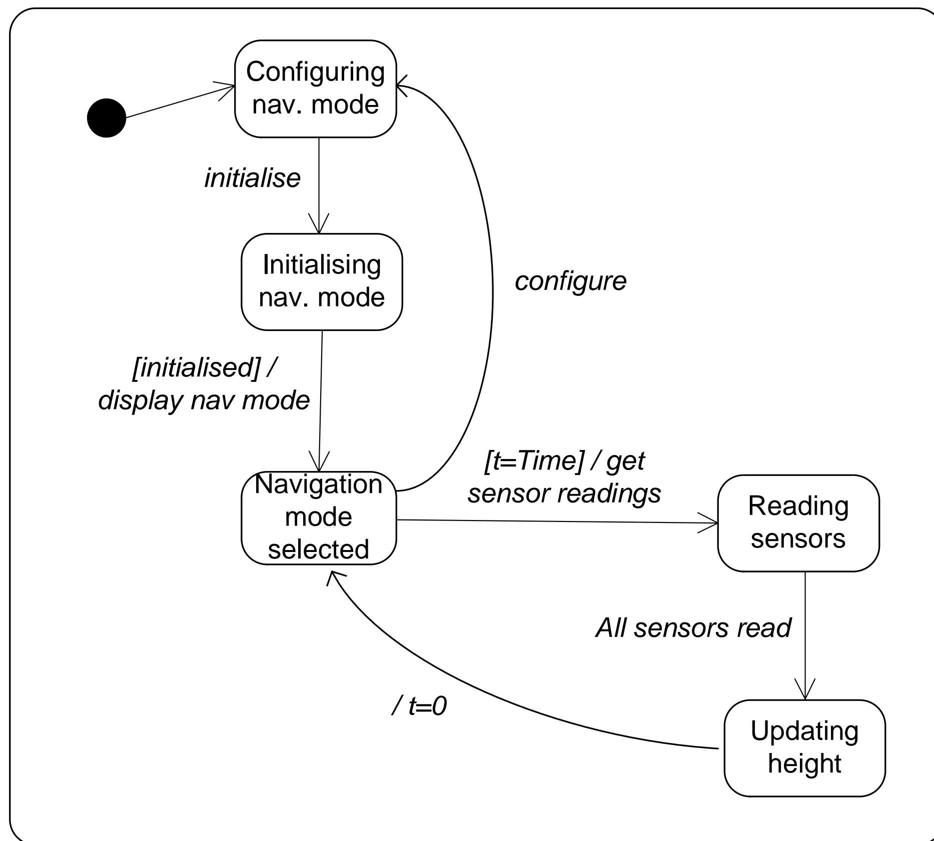


Figure D.11: Statechart model for the Navigator class

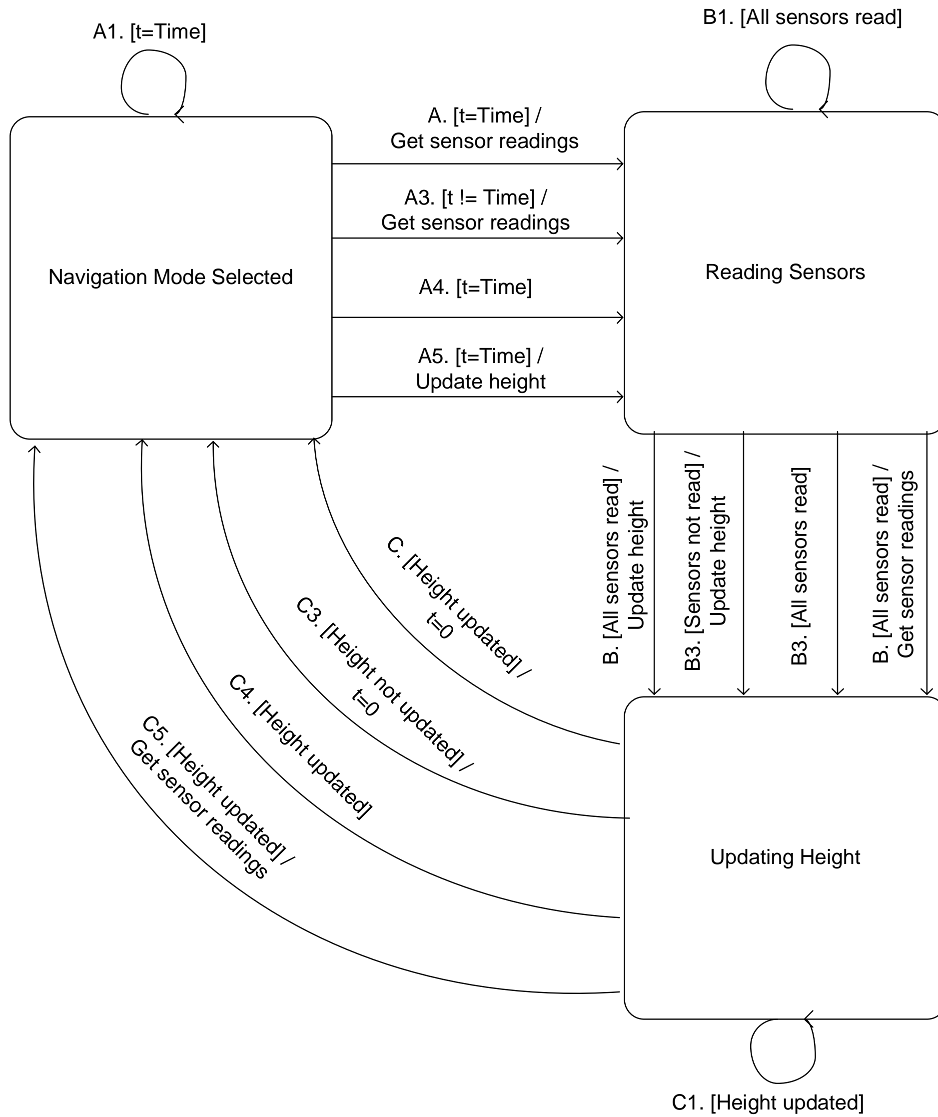


Figure D.12: Mutated statechart for the Navigator class

Task	Deviation	Effect
1		No applicable deviations. If TF mode is not enabled, hazard does not exist
2	**Quick**	Not hazardous – The minimum height should be processed as quickly as possible
	Slow	***Hazardous*** **–** Delay in changing the minimum height could hazardous
	Early	Not hazardous – No requirement to wait before altering the minimum height
	Late	Not hazardous – Minimum height will only be altered as required by the pilot
3	**Quick**	Not hazardous – It is desirable that the current height is updated as quickly as possible
	Slow	***Hazardous*** **–** A delay in determining current height will delay TF cue to pilot
	Early	Not hazardous – It is desirable that there is no delay in obtaining current aircraft height
	Late	***Hazardous*** – The current height must be updated regularly to ensure TF cue remains accurate
4	**Quick**	Not hazardous – It is desirable that the TF cue is displayed as quickly as possible
	Slow	***Hazardous*** **–** A delay in the TF cue to the pilot is hazardous
	Early	Not hazardous – It is desirable that there is no delay in calculating a new radar cue
	Late	***Hazardous*** **–** The TF cue must be updated regularly

Figure D.13: Applying timing deviations to tasks

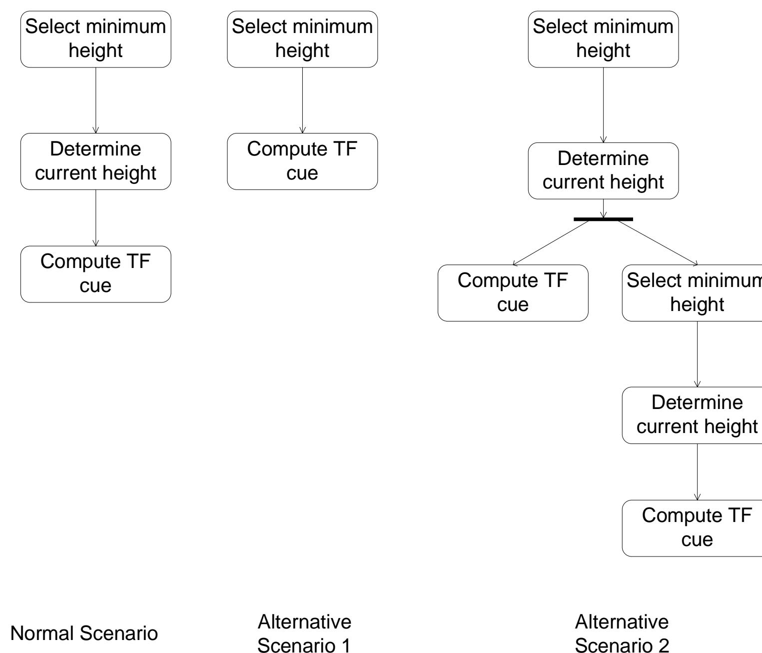


Figure D.14: Alternative scenarios

Manipulator	Deviation	Parameter	Return
Minimum_Height()	Transformer		Minimum_Height
Display_Minimum_Height()	Transformer	Minimum_Height	
Get_Height()	Transformer		Height_In_Feet
Update_Height()	Transformer	Height_In_Feet	
Calculate_TF_Radar_Cue()	Transformer		
Display_TF_Radar_Cue()	Transformer	TF_Radar_Cue	

Figure D.15: Manipulators of critical data

System: Avionics Control System			
Class: Flight Director			
Supplier Interactions	**Safety Obligations**	**Otherwise**	**Assumptions**
None			
Client Interactions			
Minimum_Height()	None		Pilot will alter minimum height when necessary
Display_TF_Radar_Cue (TF_Radar_Cue)	previousDisplay.at+1000>=Time.now **and** Param1=((Flight_Director.Minimum_Height-Flight_Director.Height_in_Feet)/Flight_Director.Minimum_Height)*100 **and** -100<=param1<=100	Param1=100	

Figure D.16: Safety obligations for the Flight Director class

System: Avionics Control System			
Class: Navigator			
Supplier Interactions	**Safety Obligations**	**Otherwise**	**Assumptions**
None			
Client Interactions			
Get_Height()	previousGet_Height.at+1000>=Time.now	Self.status=invalid	
Update_Height(Height_in_Feet)	Param1>0		

Figure D.17: Safety obligations for the Flight Director class

System: Avionics Control System			
Class: INS			
Supplier Interactions	**Safety Obligations**	**Otherwise**	**Assumptions**
Get_Height()	previousGet_Height.result-result<=8000 **and** Time.now<=Time.now@pre+50	Self.status=invalid	
Client Interactions			
None			

Figure D.18: Safety obligations for the Navigator class

System: Avionics Control System			
Class: Control Panel			
Supplier Interactions	**Safety Obligations**	**Otherwise**	**Assumptions**
Minimum_Height()	Result=voltage_input*200 **and** 0<result<=1000 **and** Time.now<=Time.now@pre+100	Result=500	
Client Interactions			
None			

Figure D.19: Safety obligations for the INS class

System: Avionics Control System			
Class: Aircraft			
Supplier Interactions	**Safety Obligations**	**Otherwise**	**Assumptions**
Update_Height()	Time.now<=Time.now@pre+50		
Client Interactions			
None			

Figure D.20: Safety obligations for the Control Panel class

System: Avionics Control System			
Class: Status Display			
Supplier Interactions	**Safety Obligations**	**Otherwise**	**Assumptions**
Display_TF_Radar_ Cue(TF_Radar_Cue)	Time.now<=Time.now@pre+100		
Client Interactions			
None			

Figure D.21: Safety obligations for the Aircraft class

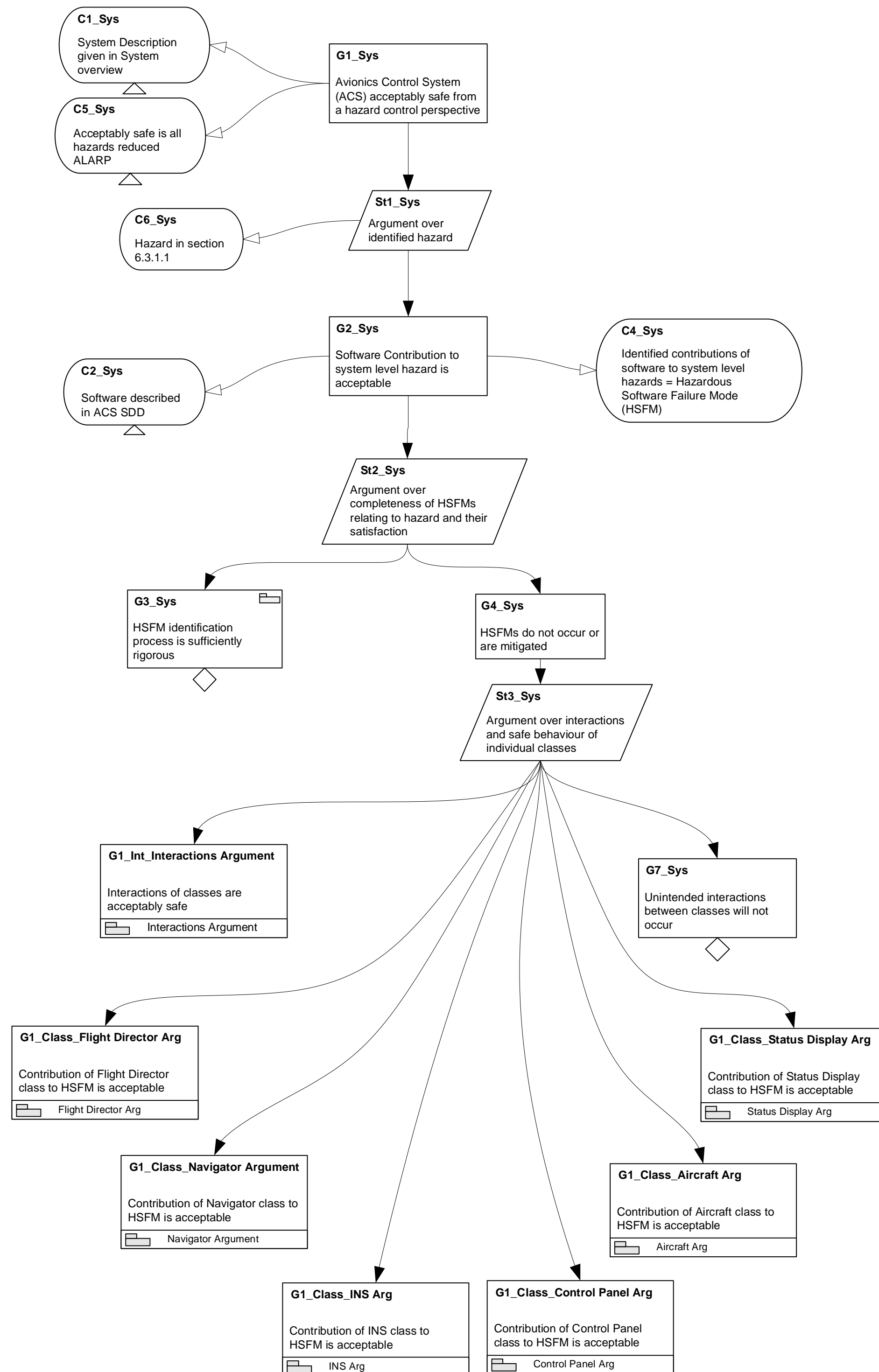


Figure D.22: Software system level argument for the ACS

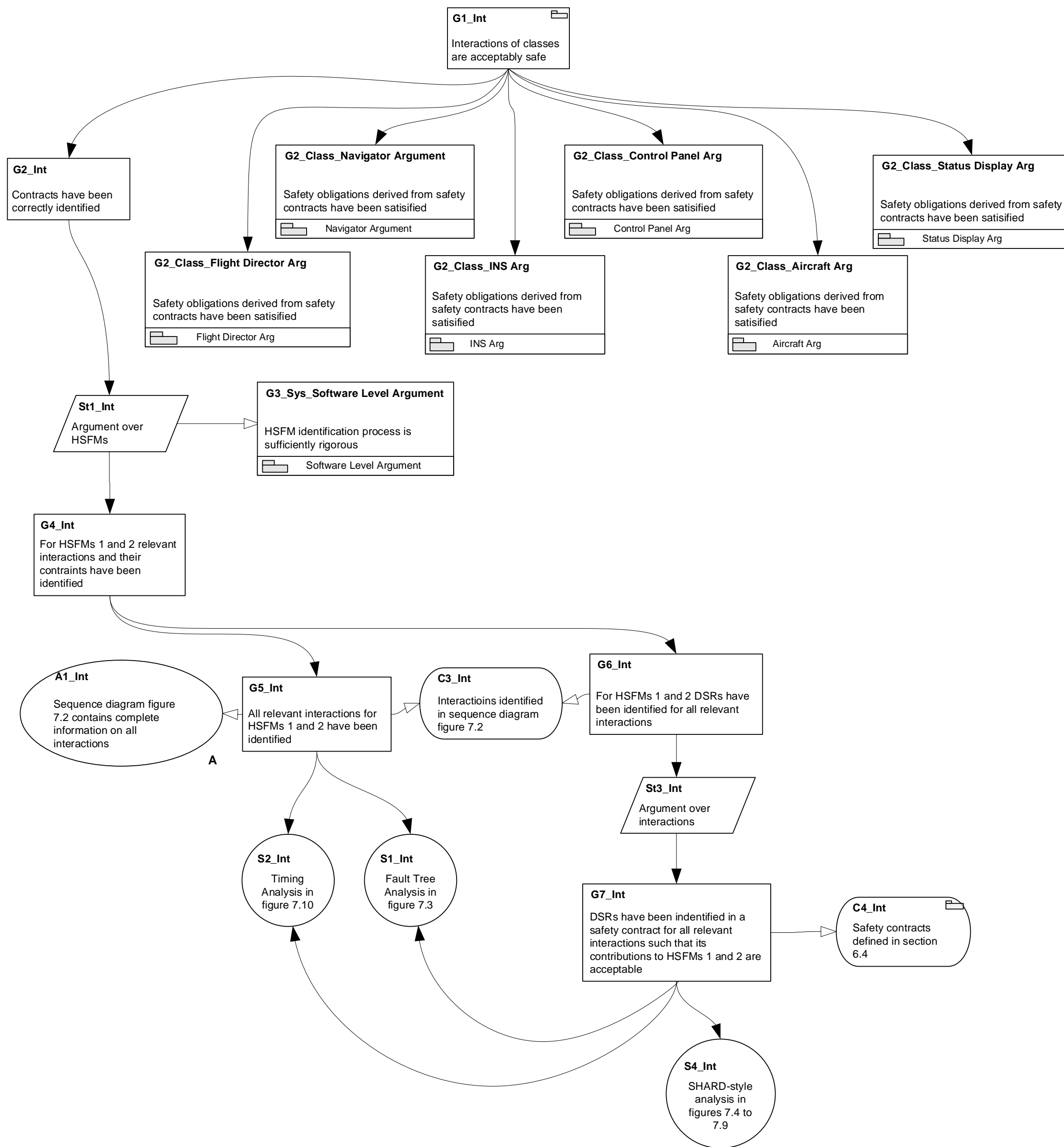


Figure D.23: Interactions argument for the ACS

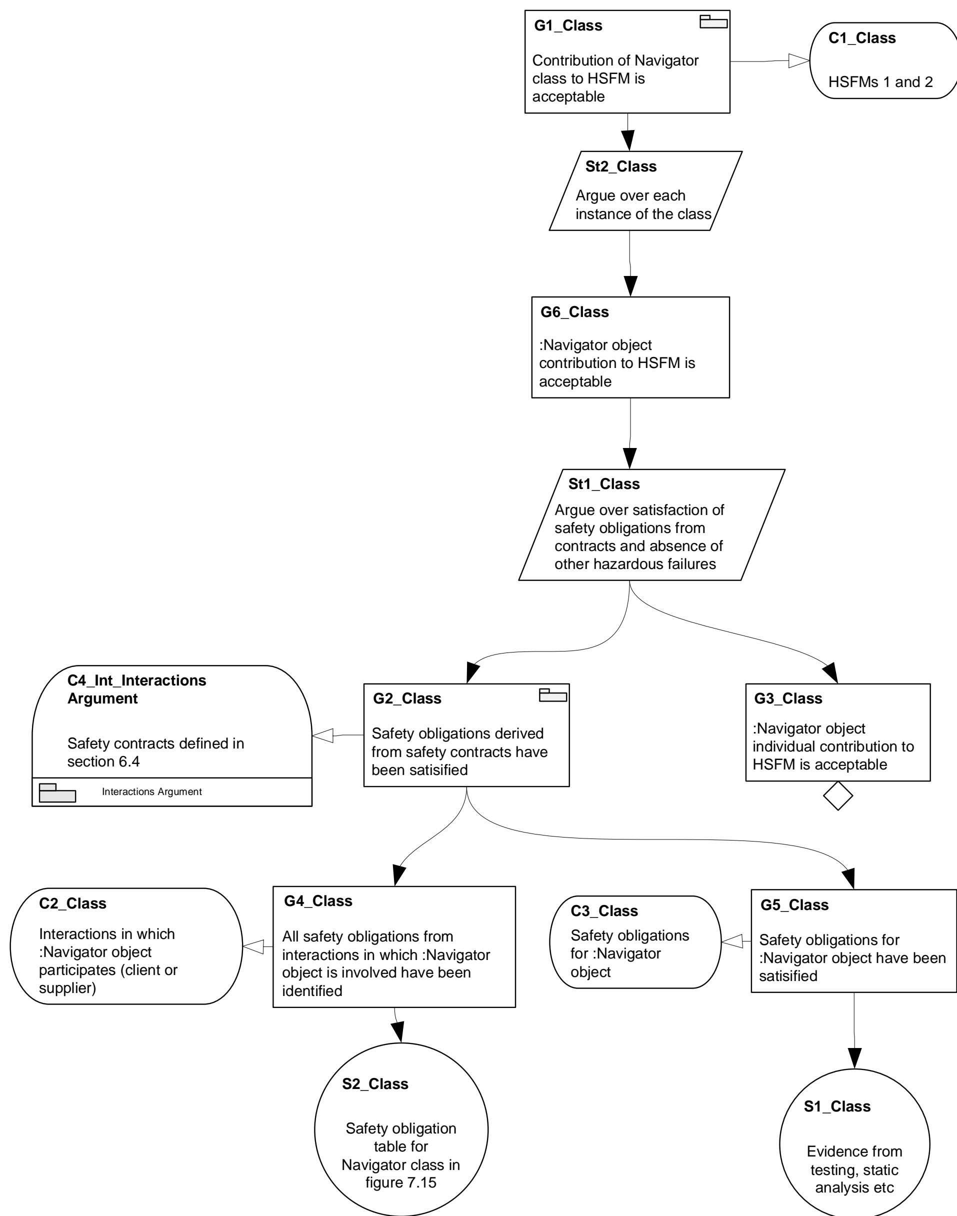


Figure D.24: Navigator class argument for the ACS

List of References

[1] K Allenby and T Kelly. Deriving safety requirements using scenarios. In *5th IEEE International Symposium on Requirements Engineering(RE'01)*. IEEE Computer Society Press, 2001.

[2] Federal Aviaition Authority. *Handbook for Object-Oriented Technology in Aviation*. Federal Aviaition Authority, 2004.

[3] Stephane Barbey and Alfred Strohmeier. The problematics of testing object-oriented software. In *SQM '94 Second Conference on Software Quality Management*, volume 2, pages 411–426, July 26-28 1994.

[4] Imran Bashir and Amrit L. Goel. *Testing Object-Oriented Software - Life Cycle Solutions*. Springer, 2000.

[5] Antoine Beugnard, Jean-Marc Jezequel, Noel Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, June, 1999:38–45, 1999.

[6] A. Bondavalli and L. Simoncini. Failure classification with respect to detection. In *First Year Report, Task B: Specification and Design for Dependability*. ESPRIT BRA Project 3092 Predictably Dependable Computing Systems, May 1990.

[7] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjammin-Cummings, 1994.

[8] Maria Cengarle and Alexander Knapp. Towards ocl/rt. *Lecture Notes in Computer Science*, 2391:390–409, 2002.

[9] CISHEC. *A guide to hazard and operability studies*. The chemical industry safety and health council of the chemical industries association Ltd., 1977.

[10] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Prentice-Hall, 1990.

[11] Derek Coleman. *Object-oriented development: the fusion method.* Prentice Hall, 1994.

[12] David Crocker. Safe object-oriented software: The verified design-by-contract paradigm. In Felix Redmill and Tom Anderson, editors, *Practical Elements of Safety - Proceedings of the Twelfth Safety-critical Systems Symposium*, February 2004.

[13] Bruce Powell Douglass. *Real-Time UML - Developing Efficient Objects for Embedded Systems.* Addison-Wesley, 1998.

[14] Bruce Powell Douglass. *Doing Hard Time - Developing Real-Time Systems with UML, Objects, Frameworks, And Patterns.* Addison-Wesley, 1999.

[15] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards and Interfaces*, 19:325–334, 1998.

[16] P.D. Ezhilchelvan and S.K. Shrivastava. A characterization of faults in systems. Technical Report CS-TR: 206, University of Newcastle upon Tyne, 1989.

[17] Janusz Gorski and Bartosz Nowicki. Object oriented approach to safety analysis. *Proc. ENCRESS '95*, pages 338–350, 1995.

[18] Janusz Gorski and Bartosz Nowicki. Safety analysis based on object-oriented modelling of critical systems. *Proc. SAFECOMP '96*, pages 46 – 60, 1996.

[19] M.J. Harrold, J.D. McGregor, and K.J. Fitzpatrick. Incremental testing of object-oriented class structure. In *14th International Conference on Software Engineering*, 1992.

[20] Richard Hawkins, Ian Toyn, and Iain Bate. An approach to designing safety critical systems using the unified modelling language. In *Critical Systems Development with UML - Proceedings of the UML'03 Workshop.* TUM, 2003.

[21] Richard Helm, Ian Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Object-Oriented Programming Systems, Languages and Applications Conference*, Special Issue of SIGPLAN Notices, pages 169–180. ACM Press, 1990.

[22] HMSO. Health and Safety at Work etc. Act, 1974.

[23] IEC. *61508 - Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems.* International Electrotechnical Commision, 1998.

[24] Ivar Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object Oriented Software Engineering - A Use Case Driven Approach.* Addison-Wesley, 1992.

[25] Jean-Marc Jezequel and Bertrand Meyer. Design by contract: The lessons of Ariane. *Computer*, pages pp. 129–130, 1997.

[26] JIMCOM. *The official handbook of Mascot version 3.1.* Joint IECCA and MUF committee on Mascot, 1987.

[27] Per Johannessen, Christian Grante, Anders Alminger, Ulrik Eklund, and Jan Torin. Hazard analysis in object oriented design of dependable systems. In *2001 International Conference on Dependable Systems and Networks (DSN 2001)*. IEEE Computer Society, 2001.

[28] J. Jürjens. Developing safety-critical systems with UML. In *Proc. UML 2003 - The Unified Modelling Language 6th International Conference*, pages 360–372, 2003.

[29] Tim Kelly. *Arguing Safety - A Systematic Approach to Managing Safety Cases*. PhD thesis, Department of Computer Science, The University of York, 1998.

[30] Tim Kelly. Concepts and principles of compositional safety case construction. Technical Report COMSA/2001/1/1, The University of York, 2001.

[31] Tim Kelly. Managing complex safety cases. In *11th Safety Critical Systems Symposium (SSS'03)*. Springer-Verlag, February 2003.

[32] Sun-Woo Kim. *Assessing the Adequacy of Test Data for Object-Oriented Programs Using the Mutation Method*. PhD thesis, Department of Computer Science, The University of York, 2001.

[33] Chenho Kung. The object-oriented paradigm. *Encyclopedia of Microcomputers*, November 1991.

[34] D. Kung, N. Suchak, P. Hsia, Y. Toyoshima, and C. Chen. On object state testing. In *Proc. of COMPSAC'94*. IEEE Computer Society Press, 1994.

[35] David Kung, Jerry Gao, and Pei Hsia. A test strategy for object-oriented programs. In *Proceedings of 19th Annual International Computer Software and Applications Conference (COMPSAC '95)*, pages 239–244, Dallas, TX, USA, August 1995.

[36] Jet Propulsion Laboratory. Software failure modes and effects analysis. *Software product assurance handbook*, 1995.

[37] Kevin Lano, David Clark, and Kelly Androutsopoulos. Safety and security analysis of object oriented models. *Lecture Notes in Computer Science*, 2434:82 – 93, 2002.

[38] N. G. Leveson and P.R. Harvey. Software fault tree analysis. *Journal of Systems and Software*, pages 173–181, 1983.

[39] N. G. Leveson and T. J. Shimeall. Safety verification of Ada programs using software fault trees. *IEEE Software*, 8:48–59, 1991.

[40] Nancy Leveson. *Safeware - System Safety and Computers*. Addison-Wesley, 1995.

[41] Jacques-Louis Lions. Ariane 5 flight 501 failure report by the inquiry board. Technical report, ESA, July 1996.

[42] Barbara Liskov and Jeanette Wing. A behavioural notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16:1811–1841, 1994.

[43] Robyn R. Lutz and Robert M. Woodhouse. Experience report: Contributions of SFMEA to requirements analysis. *ICRE '96*, 1996.

[44] James Martin and James Odell. *Object-Oriented analysis and design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[45] John McDermid. Software safety: Where's the evidence? In *Australian Workshop on Industrial Experience with Safety Critical Systems and Software*, 2001.

[46] Bertrand Meyer. Applying "design by contract". *Computer*, Oct. 1992:40–51, October 1992 1992.

[47] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[48] Richard Mitchell and Jim McKim. *Design by Contract*. Addison-Wesley, 2002.

[49] MoD. *Defence Standard 00-56 Issue 2: Safety Management Requirements for Defence Systems*. HMSO, 1996.

[50] MoD. *Defence Standard 00-58: HAZOP Studies on Systems Containing Programmable Electronics*. HMSO, 1996.

[51] MoD. *Defence Standard 00-55 Issue 2: Requirements for Safety Related Software in Defence Equipment*. HMSO, 1997.

[52] MoD. *Interim Defence Standard 00-56 Issue 3: Safety Management Requirements for Defence Systems*. HMSO, 2004.

[53] Gail C. Murphy, Paul Townsend, and Pok Sze Wong. Experiments with cluster and class. *Communications of the ACM*, 37(9), 1994.

[54] Bartosz Nowicki and Janusz Gorski. Object oriented safety analysis of an extra high voltage substation bay. *Lecture Notes in Computer Science*, 1516:306–315, 1998.

[55] Society of Automotive Engineers Inc. Aerospace Recommended Practice (ARP) 4754: Certification considerations for highly-integrated or complex aircraft systems, November 1996.

[56] Society of Automotive Engineers Inc. Aerospace Recommended Practice (ARP) 4761: Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment, December 1996.

[57] United States Department of Defense. *MIL-STD-882C: System Safety Program Requirements*. US Department of Defense, 1993.

[58] Office of Nuclear Regulatory Research. Fault tree handbook. U.S. Nuclear Regulatory Commission, January 1981.

[59] OMG. *Unified Modelling Language Specification version 1.4*. Object Management Group, 2001.

[60] OMG. UML 2.0 OCL specification. Technical Report ptc-03-10-14, Object Management Group, 2003.

[61] OMG. Unified modeling language: Superstructure, version 2.0. OMG, August 2005.

[62] Dewayne E. Perry and Gail E. Kaiser. Object-oriented programs and testing. *The Journal of Object Oriented Programming*, 1990.

[63] R. J. Pooley. *An Introduction to Programming in SIMULA*. Blackwell Scientific Publications, Oxford, 1987.

[64] David Pumfrey. *The Principled Design of Computer System Safety Analyses*. PhD thesis, Department of Computer Science, The University of York, 1999.

[65] D. J. Reifer. Software failure modes and effects analysis. *IEEE Transactions on reliability*, 28(3), 1979.

[66] RTCA. *DO-178B - Software Considerations in Airborne Systems and Equipment Certification*. Radio and Technical Commission for Aeronautics, 1992.

[67] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modelling and Design*. Prentice Hall, 1991.

[68] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.

[69] Chris Sampson. Evidence gathering using static code analysis. In Felix Redmill and Tom Anderson, editors, *Practical Elements of Safety - Proceedings of the Twelfth Safety-critical Systems Symposium*. Springer-Verlag, February 2004.

[70] Sally Shlaer and Stephen Mellor. *Object-oriented systems analysis: modelling the world in data*. Yourdon press, 1988.

[71] Anthony Simons and Ian Graham. 30 things that can go wrong in object modelling with UML 1.3. In H Kilov, B Rumpe, and I Simmonds, editors, *Behavioural Specifications of Businesses and Systems*, pages 237–257. Kluwer Academic Publishers, 1999.

[72] John Simpson and Edmund Weiner, editors. *Oxford English Dictionary*. Oxford University Press, second edition edition, 1989.

[73] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. *Proceedings of OOPSLA '96*, 31(10):268–285, 1996.

[74] Neil Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[75] Francis Thom. Safety in the loop: An overview of system safety issues throughout the product development lifecycle. Technical report, ARTiSAN Software Tools, 2002.

[76] Phil Thornton. Software Design Description (SDD) for the avionics control system. Technical Report BAE-WSC-SD-GEN-PRD-0108, BAE Systems, 2003.

[77] Phil Thornton. Software Requuirements Specification (SRS) for the avionics control system. Technical Report BAE-WCSC-SR-GEN-PRD-0107, BAE Systems, 2003.

[78] Tatsuhiro Tsuchiya, Hirofumi Terada, Shinji Kusumoto, Tohru Kikuno, and Eun Mi Kim. Derivation of safety requirements for safety analysis fo object-oriented design documents. In *21st International Computer Software and Applications Conference (COMPSAC '97)*. IEEE Computer Society, 1997.

[79] C.D. Turner and D.J. Robson. State-based testing and inheritance. Technical Report 1/93, University of Durham, 1993.

[80] Department of Computer Science University of York. CAS: Computers and software and ISA. Course Notes of MSc in Safety Critical Systems Engineering, April 2004.

[81] Alain Villemeur. *Reliability, availability, maintainability, and safety assessment*, volume 1. John Wiley and Sons, 1992.

[82] Jos Warmer and Jos Kleppe. *The Object Constraint Language - Precise Modelling with UML*. Addison-Wesley, 1999.

[83] R. Weaver, J. McDermid, and T. Kelly. Software safety arguments: Towards a systematic categorisation of evidence. In *20th International System Safety Conference*, 2002.

[84] R. A. Weaver. *The safety of Software - Constructing and Assuring Arguments*. PhD thesis, Department of Computer Science, The University of York, 2003.

[85] Elaine J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, 12(12):1128–1138, December 1986.

[86] S.A. Whitford. Software safety code analysis of an embedded C++ application. In *20th International System Safety Conference*, 2002.

[87] Ken Wong. Deriving design criteria for safety-critical object-oriented software systems. In *Proceedings of the 21st International System Safety Conference*, 2003.