Structuring Diversely Designed Software

Wanderley Lobianco Júnior

March, 1994

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY COLLEGE LONDON
UNIVERSITY OF LONDON

This thesis is submitted to the University of London in partial fulfilment of the requirements for the degree of Doctor of Philosophy (Ph.D.)
Abstract

There are a number of computer control systems in which a fault in the software can endanger human lives or lead to substantial financial loss. Software fault-tolerance provides tolerance to residual design faults in the software by means of diversely designed versions which comply with a single specification. This has proved to be an effective way of increasing software reliability. However, there are designers who hesitate to adopt this approach because of the additional cost of producing and integrating diversely designed software and assessment algorithms. Hence, guidelines are needed for structuring fault-tolerant software so that the time and cost of developing and integrating multiple versions and assessment components are reduced.

The main contribution of this research is to propose and justify a framework for structuring fault-tolerant software based on reusable components that encapsulate data and communicate by message exchange via indirect addressing. Reusable components allow for a reduction in the cost of system development, as fewer specially tailored modules are needed. Moreover, the reliability of reusable components have been observed in the field.

The framework provides a transparent filter between the diversely designed versions and the modules that communicate with them. Therefore, fault-tolerant software modules are interchangeable with functionally equivalent components that do not tolerate residual design faults. As a result, software components do not need to have special interfaces either to communicate with fault-tolerant modules or to be configured as diversely designed versions.

The framework caters for the realisation of fault-tolerant modules based on well-known software fault-tolerance techniques, such as: compensation, exception handling, recovery blocks, and n-version programming. In addition, it extends their applicability by providing a structured way of combining them.
To

my wife Terezinha,

my parents Wanderley and Nelma,

my grandparents Quito and Dinah,

from whom I have received so much love and support.
Acknowledgements

I am deeply grateful to my supervisor, Professor Steve R. Wilbur, for his guidance and advice. His support has been fundamental to my research. His comments and his ideas have been of great value throughout the development of this work.

I am thankful to Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq, grant No. 202167/89-5) for their financial support. In particular, I would like to thank Dr. Nelson Prugner for being always very helpful.

Special thanks are due to Ben Bacarisse, Tony Ballardie, Stephen Hailes, David Lee, David Parrott, Mian Wei, and Russel Winder, whose comments and suggestions have contributed significantly to improve this work.

May I also express my gratitude to Air-Chief-Marshals Socrates Monteiro, ESCA Computing Systems (Brazil), and the Civil Aviation Authority-CAA (U.K.) for their valuable contributions to the experiment in air traffic control carried out in this research. I wish to thank Dr. Stephen Hailes for his co-supervision of the project and J. Asiedu, A. Dove, L. Du, C. Harris, A. Mahmood, and N. Stchedroff for their competent work in the specification, design, implementation, and documentation of the system.

To my parents Wanderley and Nelma, and to my brother, Luís Eduardo, I will be eternally in debt for their love, altruism, incentive, and dedication. I could not have grown up in a better environment than the one they provided me with.

To my parents-in-law, Mario and Ivone, I wish to thank not only for their love and support, but also for having given me the most important person in my life, my wife Terezinha.

I wish to express my gratitude to Capello Ivo and Wanda Folhadella, who were very special to me, for all their love and care.

I am very thankful to my brother-in-law Mario Roberto, my sister-in-law Helena, and my nephew André for making us feel at home in London.

Words are too weak to express how much I own to my wife Terezinha. She has given me all support, incentive, dedication that I needed. Above all, she has given me her love, more than I could ever dream of. I wish to thank my wife for loving me so much and for staying by my side at all times in my life.

Finally, I must thank God for my life and for His blessings and guidance throughout this work.
# Contents

1 Introduction .................................................. 20
   1.1 Main Goals ................................................... 22
   1.2 Plan of the Thesis ........................................... 23

2 Review on Software Fault-Tolerance ................. 27
   2.1 Concepts and Terminology .............................. 28
   2.2 Classification of Fault-Tolerance Techniques .......... 31
   2.3 Techniques Based on Identical Replicas .......... 32
      2.3.1 Passive Replication ................................. 32
      2.3.2 Active Replication .................................. 34
   2.4 Techniques Based on Design Diversity .......... 35
      2.4.1 Compensation ......................................... 36
      2.4.2 Exception Handling .................................. 36
      2.4.3 Recovery Blocks ...................................... 38
Contents

5.6 Multi-Port SoFT Modules ......................................... 109
5.6.1 Synchronisation of the Mono-port Controllers ........ 111
5.7 Summary ................................................................. 114

6 Analysis of the SoFT Model ........................................ 117
6.1 Basic Configurations ............................................... 117
6.2 Traditional Software Fault-Tolerance Techniques in SoFT ... 126
6.2.1 Compensation ..................................................... 127
6.2.2 Exception Handling ............................................. 128
6.2.3 Recovery Blocks ............................................... 130
6.2.4 N-Version Programming ...................................... 130
6.3 Complexity of Messages in SoFT Modules ................. 132
6.3.1 Complexity of Messages at Each Controller Port ...... 132
6.3.2 Global Complexity of Messages ........................... 134
6.4 Availability Issues .................................................. 139
6.5 Testing ................................................................. 140
6.6 Summary ................................................................. 141

7 Assessment of the SoFT Controller ............................... 143
7.1 Modular Structure .................................................... 144
Contents

7.2 Implementation of the Controller ............................................. 146
7.3 Numeric Analysis ................................................................. 148
7.4 Summary .......................................................... 154

8 An Application in Air Traffic Control ................................ 155

8.1 Objectives ................................................................. 155

8.2 The Experiment ............................................................... 157
  8.2.1 Executive Overview ................................................. 157
  8.2.2 Technical Description .............................................. 158

8.3 Incorporation of Software Fault-Tolerance .................. 162
  8.3.1 Plan Validation ...................................................... 162
  8.3.2 Route Extraction .................................................... 164
  8.3.3 Approaching Lists .................................................. 165
  8.3.4 Collision Warning .................................................. 167

8.4 Conclusions of the Experiment ........................................ 169

8.5 Summary ................................................................. 170

9 Limitations and Related Work .......................................... 171

9.1 Non-Transparent Error Recovery .................................... 171
  9.1.1 Recovery Based on the Feedback of the Reliable Output ... 172
Contents

B.4.2 Sequential Servers ................................................................. 252

C POPE: a Port-Oriented Programming Environment .................. 257
  C.1 Primitive Components .......................................................... 257
  C.2 The Kernel .............................................................................. 259
  C.3 Cluster Components ............................................................... 260
    C.3.1 Composite Components .................................................... 262
    C.3.2 Communication Buses ...................................................... 263
  C.4 Primitives .............................................................................. 264
    C.4.1 Configuration Primitives .................................................. 265
    C.4.2 Communication Primitives .............................................. 268
  C.5 Examples ............................................................................... 272
  C.6 Summary .............................................................................. 279

D Principles of Air Traffic Control ............................................... 280
  D.1 Main Air Traffic Control Activities ...................................... 281
    D.1.1 Analysis of Flight Plans .................................................. 282
    D.1.2 Analysis of Radar Data ................................................... 284
  D.2 Automated Air Traffic Control ............................................. 285
Contents 13

Glossary 287

Bibliography 294
List of Figures

1.1 Illustration of a Fault-Tolerant Software Module .................................................. 22

2.1 Example of a System Structure .............................................................................. 29

2.2 Framework for an Ideal Exception-based Fault-tolerant Component .................. 37

2.3 Example of a Recovery Block ................................................................................. 39

2.4 The Domino Effect .................................................................................................. 42

2.5 Processes Transactions via Conversations ................................................................ 43

2.6 General Triple-Modular Redundancy (TMR) Diagram .......................................... 44

2.7 An Example of Two Groups Sending Messages to a Third .................................... 49

2.8 Hierarchical Representation of Software Fault-Tolerance Techniques .................. 50

3.1 Black Box Model of a Software Module ............................................................... 52

3.2 Black Box Model Classification as to its Inputs and Outputs ................................. 53

3.3 Use of Diversity in the Phases of the Software Development Process .................. 61
List of Figures

4.1 Outline of a Software Fault-Tolerant Component ................................ 67
4.2 Generic Configuration of the Assessment Subsystem ............................ 72
4.3 Diagram of a Fault-Tolerant Assessment Component ........................... 73

5.1 A SoFT Module ...................................................................................... 84
5.2 Controller Interaction with the Input Validation Module ....................... 89
5.3 Example of an Input Validation Module ............................................... 90
5.4 Input Validation Protocol ........................................................................ 91
5.5 Initiation of a New Round of Voting .................................................... 91
5.6 Voter Protocol with Static Output Assessment Order ........................... 93
5.7 Voter Protocol with Dynamic Output Assessment Order ...................... 94
5.8 Controller Communication with the Voter ............................................ 95
5.9 Example of a Majority Voter ............................................................... 96
5.10 Controller Interaction with the Conformity Test .................................. 97
5.11 Example of a Conformity Test .............................................................. 97
5.12 Output Assessment Protocols .............................................................. 99
5.13 Controller Interaction with the Compensation Modules ..................... 100
5.14 Example of a Compensation Module ................................................ 100
5.15 Controller Interaction with an Error Handling Module ..................... 102
List of Figures

5.16 Example of an Error Handling Module .................................. 102
5.17 A Client SoFT Module with a Matching Voter ...................... 109
5.18 Simplified Architecture of a Filter SoFT Module .................. 112
5.19 Controller Algorithms for Token Allocation and Release ........ 113
5.20 Algorithm of the Token Administrator .................................. 114
5.21 Mono-controllers - Token Administrator Simplified Diagram .... 115

6.1 SoFT Framework for Analysis .............................................. 118
6.2 General Structure of a Producer SoFT Module ....................... 120
6.3 High Level Specification of a Producer Controller ................ 120
6.4 General Structure of a Consumer SoFT Module ....................... 121
6.5 High Level Specification of a Consumer Controller ................. 121
6.6 General Structure of a Client SoFT Module .......................... 122
6.7 High Level Specification of a Client Controller ..................... 123
6.8 General Structure of a Server SoFT Module ......................... 124
6.9 High Level Specification of a Server Controller .................... 125
6.10 Example of a SoFT Server based on Compensation ................. 128
6.11 Example of a SoFT Server based on Exception ...................... 129
6.12 Example of a SoFT Server based on the Recovery Block Scheme .. 131
List of Figures

6.13 Example of a SoFT Server based on N-version Programming ............... 131

7.1 Module with a Monolithic Fault-Tolerance Component ...................... 144

7.2 Possible Configurations of SoFT Components ................................. 147

8.1 Diagram of the Air Traffic Control System ..................................... 159

8.2 Air Route Chart of The United Kingdom ........................................ 160

8.3 Fault-Tolerant Configuration of Module Plan .................................. 164

8.4 Fault-Tolerant Configuration of Module Scheduler ........................... 166

8.5 Fault-Tolerant Configuration of Module Collision ............................ 168

9.1 Normal and Error Recovery Codes in a Single Component .................. 173

9.2 External Assessment of the Internal State of a Module ....................... 174

9.3 A SoFT Module Assessing the Internal States of SoFT Versions .......... 176

9.4 Example of a SoFT Client based on the Recovery Block Approach ....... 180

9.5 Equivalent Server Configurations in the FTM and SoFT Models .......... 184

10.1 The SoFT Framework ................................................................. 187

10.2 Possible Configurations of SoFT Components (reproduction) ............ 191

A.1 A Server SoFT Module in a Client-Server Configuration ..................... 197

A.2 SoFT Server Σ ................................................................. 197
List of Figures

C.1 Ports and Links ......................................................... 259
C.2 Example of a Simple Client-Server Configuration in POPE ........ 261
C.3 Example of Composite Components .................................. 263
C.4 Example of Components Linked via a Communication Bus ........ 264
C.5 Diagram of a Client-Server-Alarm Configuration .................. 273
C.6 Example of a Configuration Component ............................. 274
C.7 Example of a Server Component .................................... 275
C.8 Example of a Client Component ..................................... 277
C.9 Example of a Consumer Component .................................. 278
D.1 Example of a Flight Plan Form ....................................... 283
List of Tables

5.1 Greek Letters used in SoFT Modules .............................................................. 86
5.2 Acronyms used in SoFT Modules ................................................................. 86
6.1 SoFT Configurations for Message Complexity Analysis ............................. 136
6.2 Message Complexity Analysis for $n = 3$ and $k = 0$ ................................. 138
7.1 SoFT Controller Function Sizes ................................................................. 150
7.2 SoFT Controller Function Utilisation ......................................................... 151
7.3 Sizes of the Possible Configurations of the SoFT Controller ............... 152
7.4 Specific SoFT Controllers Grouped by Role and Version Execution Mode ................................................................. 153
Chapter 1

Introduction

There are a number of critical applications where a failure in the computer control system can endanger human lives or lead to substantial financial loss. For a long time, fault-tolerance techniques have been used to cope with hardware faults [Car87]. More recently, designers have also been concerned with residual faults in the software which manifest themselves only when the systems are already in operation. Railways, nuclear reactors, and aircraft are amongst the areas in which design diversity has been used to develop reliable software [Vog88].

All software has faults and limitations, despite application of software engineering techniques. Although it is quite reasonable to assume that software systems will not be released with known faults, software faults can be responsible for up to one quarter of all system failures [BGH87, CG87]. To test a single and simple sequential program can be relatively easy. However, in concurrent systems, it is impracticable to test all valid simultaneous states of the processes that compose a large system. Thus, inadvertent faults may persist after extensive testing. Since computing systems are becoming increasingly more important in the control of critical processes, it is imperative that their reliability be enhanced not only through the utilisation of fault-avoidance techniques, but also by means of fault-tolerance techniques.
All forms of fault-tolerance are based on redundancy [LKA88]. Redundant components perform equivalent functions which are assessed by special algorithms. Figure 1.1 depicts a black box model of a software module in part a and its fault-tolerant configuration in part b. In the diagram of figure 1.1.b, the computation of modules $\text{version}_1$ to $\text{version}_n$ is assessed by module $\text{assess}$. Assessment algorithms are to be significantly more reliable than the versions that they assess, because they are single points of failure. In distributed systems, assessment components are to be replicated over different machines in order to survive hardware crashes.

Physical faults can be tolerated by utilising exact replicas of hardware devices. However, as software does not degrade with time, errors in the software are always attributed to design faults\(^1\). Therefore, software fault-tolerance can only be achieved by means of redundant components which have been designed according to different strategies. In order to increase diversity, each of the redundant components should be designed by an independent team. Thus, producing diversely designed programs and special algorithms to assess them is, in general, more costly than developing its single-version counterpart, particularly when library modules cannot be utilised. This has been the major reason for the scepticism of some designers who hesitate to adopt a fault-tolerance approach for developing highly reliable systems.

This research investigates ways of structuring software fault-tolerance in order to reduce the cost of developing and integrating additional versions and assessment algorithms, without jeopardising the improvement on the degree of reliability and availability of the software system achieved by the use of software fault-tolerance techniques.

\(^1\)In this thesis, all stages in the development of a software system, from the interpretation of the specification to the implementation of the code, are generically called \textit{design}. In addition, unless stated to the contrary, the term "fault-tolerance" and its variants, e.g., "fault-tolerant", refer to \textit{software} fault-tolerance.
Chapter 1. Introduction

1.1 Main Goals

Software engineering techniques have demonstrated the advantage of designing software modules which can be used in various applications. If the same approach is to be followed in the construction of systems tolerant to software faults, it is mandatory that the design of the diversely designed versions do not depend on the algorithms that will assess them. Ideally, diversely designed versions should not even be restricted to use in fault-tolerant applications. Therefore, one of the goals of this research is to propose a means by which fault-tolerance can be externally attached to modules that have been originally designed without such concern. Another aim of this work is to investigate structures for assessment modules so that they do not need to be customised to each different application.

The rationale of the diversely designed versions and the assessment algorithms of a software fault-tolerant module differ according to the application. However, a study of existing software fault-tolerance techniques (chapter 2) shows that the rules that co-ordinate the execution and interaction of the versions and assessment components do not depend on the syntax and semantics of the messages that they exchange. This suggests the hypothesis that a small number of algorithms would
Chapter 1. Introduction

suffice to control the communication amongst versions, assessment modules, and other components of the system. This hypothesis will be examined. The fewer the number of co-ordination algorithms, the more reusable they are. Therefore, the more reliable they tend to be. Moreover, re-using the same co-ordination algorithm in distinct applications, based on different fault-tolerance approaches, avoids the inherent dangers of developing new code for every new system.

The principal contribution of this thesis is to provide a framework and guidelines for structuring fault-tolerant software based on reusable components. This has two major advantages: firstly, it reduces the cost and time involved in the development of specially tailored modules for new applications and, secondly, and more important, it allows for the use of software modules whose degrees of reliability have already been observed in the field.

1.2 Plan of the Thesis

This chapter has introduced the dissertation by presenting the motivation for the work and the goals that it proposes to achieve. The remainder of this dissertation is arranged as follows.

Chapter 2 introduces the reader to the area of software fault-tolerance. Well-known software fault-tolerance techniques are described and analysed. The purpose of this chapter is twofold: for the non-expert it provides a tutorial which will help him/her to get acquainted with the subject; for the expert it is a means of normalising terms and concepts that will be used throughout the dissertation.

Chapter 3 focuses on the nature and development process of software components. The difficulty of quantifying software reliability is commented and the factors that might be responsible for the occurrence of faults in software systems are identified. Finally, it is explained the role of diversity in the various phases of the software
Chapter 1. Introduction

development process and the importance of selective and transparent introduction of fault-tolerance to software.

Chapter 4 contains a study of the requirements of fault-tolerant software components. An analysis of the different methods of obtaining the outputs of each of the diversely designed versions, the variant types of assessment techniques, and the forms of error handling is done in this chapter. This study results on the identification of the components which are needed on a comprehensive framework for building software fault-tolerant modules.

Chapter 5 proposes a comprehensive framework for building software fault-tolerant modules, named SoFT. Firstly, the principles of programming paradigms that are suitable for designing and implementing fault-tolerant software modules are examined. Then, the generic configuration of the framework is described, followed by the protocols used by its components to interact with one another. In addition, it is shown the means by which errors in the time domain are detected and dealt with. The client configuration of the SoFT model is investigated in order to guarantee that all versions that produce a correct request within an acceptable deadline receive the reply issued by an external server. Finally, it is shown how the SoFT model can be extended in order to deal with versions that communicate through multiple ports.

Chapter 6 presents the analysis of the SoFT model. Specific configurations of the framework in which the fault-tolerant module acts as producer, consumer, client, and server are exemplified and the capability of the SoFT framework to realise well-known software fault-tolerance techniques is demonstrated. This is followed by the analysis of the complexity of the messages issued in an interaction between a SoFT module and an external module. The need for providing appropriate mechanisms for ensuring that SoFT modules survive hardware crashes is, then, highlighted. Finally, it is illustrated the use of the SoFT model as a framework for testing diversely designed versions, assessment procedures, and error handling modules.
Chapter 7 concentrates on the generic SoFT controller which has been designed and implemented as part of the research. The aim of the chapter is to present the assessment of the hypothesis that a reduced number of controllers suffice to co-ordinate all valid configurations of the components of the SoFT model.

Chapter 8 is dedicated to the description and discussion of an application in the area of Air Traffic Control. The project has set out with the aim of assessing the suitability of the SoFT framework to build fault-tolerant software modules from diversely designed components which have not been developed to be configured in such a framework.

In chapter 9 the limitations of the model are examined, indicating situations where transparency has to be abandoned and where the model cannot be used at all. In addition, the SoFT model is compared with other approaches in the area of fault-tolerant software.

Chapter 10 summarises the main achievements of the research and proposes some topics which deserve further investigation.

Appendix A contains the specification of the SoFT framework. The main purpose of this appendix is to define the algorithms and interfaces of the components of the SoFT model in an abstract notation. Most of the components specified there are application dependent and, therefore, can only be seen as examples. However, in some of these components, the interfaces are fixed.

In appendix B, the number of messages exchanged between external modules and SoFT modules is evaluated for each different configuration of the SoFT model, in the best and in the worst cases. The result of this evaluation is summarised in section 6.3.1.

Appendix C describes the programming environment for distributed applications used in the thesis. In this environment, components encapsulate data and commu-
Chapter 1. Introduction

nicate by message exchange. Addressing is indirect via ports. Components can be clustered in order to create higher level abstractions.

Appendix D introduces the principles of air traffic control. The purpose of this appendix is to aid the understanding of the functions of the components of the application described in chapter 8.
Chapter 2

Review on Software Fault-Tolerance

Fault-tolerant computing is an area which has been studied since von Newmann's work in the fifties [Ran87]. However, in the early stages, the relative high rate of failures in physical components biased researchers towards hardware faults caused by component decay. It was not before the early seventies that researchers began to be concerned with software faults.

Software engineering techniques [PvSK90] have been used to avoid or eliminate faults during the development of software systems. However, experience has shown that, although fault-avoidance techniques can be beneficial, a reduction in the incidence of faults and not their complete elimination is all that can be expected [AK81]. Therefore, it is crucial that mechanisms for tolerating residual design faults be available in critical applications. Fault-avoidance and fault-tolerance are complementary rather than competitive approaches to system reliability [Avi76, Lap85].

Concepts and terms frequently used in fault-tolerant computing are introduced in this chapter. In addition, this chapter contains a survey on well-known software fault-tolerance techniques and a description of the problems that arise in developing fault-tolerant software in distributed systems.
2.1 Concepts and Terminology

One of the most used terms in Computer Science is *system*. A *system* consists of a set of components that interact according to a design. A *component* is simply a smaller system [AL81]. These recursive definitions of system and component are appropriate to represent hierarchy. A component is said to be *primitive* if it cannot be divided into sub-components that still keep the same type of interface and *composite* otherwise. A system is constructed by interconnecting component interfaces.

The set of observable values that influence the behaviour of a system is known as *state*. The internal state of a system is described by the values of its variables and registers. A system interacts with an *environment*. The system reacts to the inputs received from the environment with a state transition and, in many cases, produces an output. The external behaviour of a system (i.e., the way it is seen by the environment) is described by a set of *external states*. The environment is a system too, possibly of a different nature.

Figure 2.1 depicts the structure of a system with five components, namely: $I$, $J$, $K$, $L$, and $M$. Each of these components can be either primitive or composite. The system has two entry points, $a$, and $b$, and two exit points, $y$, and $z$. The thick lines denote the system seen by the environment. The thin lines show the internal composition of the system, which is invisible to the environment. Similarly, the internal structure of components $I$ to $M$ is not visible to the system.

When the external state of a system is not the expected one, it is said that a *failure* has occurred, i.e., that the behaviour of the system has deviated from its specification. A failure is caused by one or more errors in the system. An *error* is defined as an incorrect internal state of the system. Errors are manifestations of defects in the system which are called *faults*. 
Chapter 2. Review on Software Fault-Tolerance

Figure 2.1: Example of a System Structure

Briefly:

- **Fault** is a defect in the system.
- **Error** is the manifestation of a fault.
- **Failure** is a deviation from the functional specification of the system.

According to Laprie [Lap85], fault-tolerance is "to provide, by redundancy, a service complying with the specification in spite of faults having occurred or occurring" in a system. Every system has a theoretical or measured probability of failing either due to component decay or to improper design. This probability leads to the definition of two terms which have been widely used in the literature [Kop79, AL82, Lap85], namely: **reliability** and **availability**.

- **Reliability**: the degree of confidence in the continuous correct behaviour of the system. In mathematical terms, it is represented as the function $R(t)$ that describes the probability that the system will not have failed by time $t$.

- **Availability**: the degree of confidence in the correct behaviour of the system at a particular time. In mathematical terms, it has the form of the function
Chapter 2. Review on Software Fault-Tolerance

\[ A(t) \] which describes the probability that the system will be operating according to the specification at time \( t \), regardless of whether it has failed previously or not.

A highly available system does not necessarily mean a highly reliable one. A system which fails relatively often has a low degree of reliability. The same system can be regarded as highly available if the average failure period is much smaller than the average time interval of correct operation.

When an output is expected, faults in the system may manifest themselves in three different ways [CASD85]:

- **omission error**: the system fails completely to produce the output. It may be caused either by a component shut down or by an algorithmic fault which prevents the program from reaching the instruction where the output is issued.

- **timing error**: the system issues the output too late or too soon. Errors in this class are usually due to system overload (unacceptable delay), or to clock out of tune (too fast or too slow).

- **byzantine error\(^1\)**: the system generates an output with an incorrect value or produces an output when no output is expected.

Powell [Pow87] denominates both omission faults (which he calls *fail-stop* faults) and timing faults as *faults in the time domain*, whereas byzantine faults are classified as *faults in the value domain*.

\[^1\]This class of errors is named after the Byzantine Generals problem [LSP82], in which all loyal generals of a Byzantine division must agree on the same plan of attack, regardless of how traitorous generals behave.
2.2 Classification of Fault-Tolerance Techniques

Hardware failures jeopardise the execution of software modules. Thus, if a software component is essential for the correct behaviour of the system, the designer must adopt a mechanism that tolerates hardware faults in the node where the software component executes. Since every fault-tolerance mechanism is based on some form of redundancy [LKA88], exact copies of the software module executing at different nodes can be utilised in order to avoid disruption of services due to hardware faults. Such mechanisms, however, will fail to cope with errors caused by faults in the design of software components. The only way of increasing the probability of continuous correct behaviour of the module, despite software faults, is to use redundant modules which are based on the same specification, but which follow different design strategies. Fault-tolerance techniques are, then, divided into:

- techniques based on identically designed replicas
- techniques based on diversely designed replicas

When an error is detected, the system may recover from it by moving either forward to a new state or backward to a previous, supposedly correct, state. Another approach to software fault-tolerance is to mask the error by utilising the outputs of diversely designed versions of the same module to produce a reliable output. Hence, fault-tolerance techniques are classified according to the approach used to deal with errors as [RLT78]:

- Error Detection and Recovery techniques, which are subdivided into:
  - Forward Error Recovery
  - Backward Error Recovery
- Error masking techniques
Forward error recovery mechanisms tend to be more efficient in dealing with expected faults (e.g., division by zero operations, malfunction of physical devices, absence of free buffers), because such mechanisms allow the designer to specify in advance the best form of coping with them. Conversely, as it is not possible to know beforehand how to cope with unexpected faults, the only alternative is either to mask the error or to rollback the system to a previous reliable state and to redo the computation using a diversely designed algorithm. For this reason, error masking and backward error recovery techniques [CR86] are used to cope with unexpected faults.

2.3 Techniques Based on Identical Replicas

The most common method of providing continuous operation of software modules despite hardware failures is to replicate them in different nodes of the network. Modules can be replicated either passively [LK86] or actively [AGK'85]. In the former, only one of the replicas executes whilst the others are kept as stand-by modules. In the latter, all replicas perform their computation simultaneously. Real-time applications, which have time constraints to satisfy, must opt for active replication if the recovery time of passive replication techniques is too long to be acceptable.

2.3.1 Passive Replication

Two distinct forms of stand-by replicas are used to prevent hardware faults from making software modules unavailable: cold stand-by and hot stand-by replicas [LK86]. The choice between them will depend on the knowledge of past events required by the application in future computations and on its real-time constraints.
Cold Stand-by

In the cold stand-by approach, when the primary module fails, a replica is created in order to take its place. The replica starts its execution from the very beginning. Thus, all previous computation of the former primary replica which has not been saved on stable storage devices that can be accessed after the crash is lost. The overhead of creating and starting the new replica and re-configuring the system can be reduced by creating the stand-by replica in advance and leaving it in the idle state until it is needed.

The cold stand-by technique is not appropriate to history-dependent (or state-retention) software modules [Ani89]. However, it is a very simple error recovery mechanism and must be strongly considered in history-independent (or stateless) applications.

Hot Stand-by

In systems where previous states of the computation influence new outputs, the stand-by replicas must keep track of the computation performed by the primary module. Hot stand-by replication is based on the copy of the current state of the primary module to the replicas at particular points called checkpoints. When an error is detected in the primary module, one of the hot stand-by replicas is elected to replace it. The new primary module starts its computation from the last checkpoint.

Checkpoint positioning is an important issue in hot stand-by mechanisms. One of the functions of checkpoints is to limit the amount of time taken by the newly elected primary replica to reach the point where the former primary module failed. One commonly used policy is to execute checkpoint calls at regular time intervals. The time interval is chosen according to the application. Periodical checkpoints need to be complemented with extra checkpoints when values that have been produced
by the primary module, or received from other modules, cannot be reproduced or obtained by the stand-by replicas [Bec92]. Checkpoints can either be explicitly positioned at particular places in the code or be automatically set [LK86].

2.3.2 Active Replication

Stand-by replicas take some time to reach the point at which the primary module has failed. In applications where this delay cannot be tolerated, active replication [AGK+85] can be used instead. In this method, all replicas execute simultaneously in different machines. Active replication can be hierarchical or non-hierarchical. In the former, only one of the replicas interacts with the rest of the system. The remaining ones receive all information addressed to the primary replica, perform the computation, but do not export any result. In the latter type, there is no primary replica and, therefore, all replicas export their results. The responsibility of deciding whether a result is new or a copy, is left to the components which communicate with them. Hierarchical active replication reduces the number of messages transmitted, but it requires algorithms for detecting the failure of the primary replica and for electing a new primary module.

Periodical checkpointing is not necessary in active replication. However, compulsory checkpoints are required for the same reasons that they were necessary in passive replication mechanisms.

Cold stand-by replication is the least costly of the fault-tolerance techniques based on replicas with identical design but, at the same time, it is the one which provides the slowest recovery strategy. Active replication is the most responsive one. However, it consumes more computational resources, such as CPU, than the hot and cold stand-by techniques, because active replicas execute permanently, whereas stand-by replicas occupy resources only if they need to be activated. Hot stand-by replication lies between the two in terms of cost-performance.
Design faults do not necessarily manifest themselves every time a program executes. Those which do are removed at the testing phase. Errors occur when particular sets of inputs cause the execution of parts of the code which contain faults. Since the fault is permanent, the error will re-occur if the same module rolls back and executes the same computation under the same conditions. Faults in the software design can only be tolerated either if the computation is re-done by a distinct software module which has been designed from the same specification, but following a dissimilar algorithm, or if the error is masked by executing a number of diversely designed modules that comply with the same specification. Achieving fault-tolerance by means of design diversity is not a new concept, as it can be noticed from the following quotation from Charles Babbage’s work dating from 1837 [Bab74]:

“When the formula to be computed is very complicated, it may be algebraically arranged for computation in two or more totally distinct ways, and two or more sets of cards may be made. If the same constants are now employed with each set, and if under these circumstances the results agree, we may then be quite secure of the accuracy of them all.”

Although design diversity does not ensure the absolute correctness of the results, because there may be coincident failure modes [KL86], it produces results that are more reliable than the ones issued by a single version. This section describes two commonly used forward error recovery techniques, namely compensation and exception handling, and three well-known fault-tolerance techniques used for coping with unexpected errors: recovery block, conversation, and $N$-version programming. Although compensation and exception handling are not exactly fault-tolerance techniques based on replication, they have been classified as techniques based on design diversity because they utilise specially designed pieces of code, which are dissimilar from the normal code, to provide fault-tolerance.
2.4.1 Compensation

Some errors are detected only after their effects have been noticed by the environment or by other modules. Therefore, it is necessary that such effects be rectified. *Compensation* [RLT78] is a forward error recovery technique in which a supplementary computation is executed in order to cancel or correct a wrong result which has been issued previously. When an error is detected, the module itself, or another one on its behalf, issues another output which amends the effect of the original one. When the faulty module retains state, external compensation must be complemented by internal recovery, otherwise the error is likely to affect the future external behaviour of the module. Usually, the utilisation of software modules that retain state in fault-tolerant configurations is not transparent (chapter 9).

Compensation algorithms cannot, however, be used in situations where the wrong output has already caused an irreversible action (e.g., the ejection of the pilot's seat from an airplane, or the detonation of an explosive). In addition, this approach is inappropriate in cases where it is very difficult or expensive to correct an erroneous action (e.g., to intercept or to destroy a missile which has been incorrectly launched). However, compensation is a simple forward error recovery technique which, if applied in time, must be considered in applications that do not require a high degree of reliability and where an incorrect result can be easily amended (e.g., reading of meteorological devices, update of information displays, etc.).

2.4.2 Exception Handling

A number of abnormal, sometimes very rare, situations can occur during the execution of a program. One way of dealing with them is to program *if-then-else* type clauses in the code. This approach, however, enlarges the program and makes it less clear. Besides, errors such as division by zero, overflow, and underflow can happen in so many places that the use of *if-then-else* type clauses is inelegant and error-prone.
Chapter 2. Review on Software Fault-Tolerance

An alternative form of recovering from foreseeable errors is to write the algorithm as if no error of this kind would ever occur and, then, provide a test mechanism at lower levels. Should an error be detected, an exception is raised and the error is treated by a separate module, exclusively designed to deal with this exceptional situation. This technique is known as exception handling [AL81, Cri82, CR86, PW90, Iss92].

Exceptions can be raised by the application module, the operating system, or by hardware components. Figure 2.2 depicts an ideal component based on exception handling. The left side of the figure illustrates the normal behaviour of the component: it receives requests from the upper level, deals with them, and sends the answers back to the upper level. If the component relies on lower level modules to execute at least part of the request, it re-sends the request, or part of it, to the lower level and, then, waits for the response.

Figure 2.2: Framework for an Ideal Exception-based Fault-tolerant Component

[Diagram of component with normal and abnormal activity paths, showing exception handling mechanisms]
When a request is received from the upper level, it is verified and, if it is not a valid input, an interface exception is signalled back to the upper level. Similarly, the component will receive an interface signal from the lower level, if it has passed an incorrect set of parameters to it.

If an error is detected during the normal computation, an exception is raised and the execution flow is transferred to the right side of the figure. The exception handler tries to recover from the error. If it succeeds, the normal execution flow is resumed. If it fails, the module reports the failure to the upper level by sending a failure exception signal. An analogous procedure is followed when unrecovered errors are reported by the lower level.

2.4.3 Recovery Blocks

Recovery blocks [Ran75] allow structuring of diversely designed versions of a module in order to achieve fault-tolerance. These versions, called alternatives, are invoked one by one, according to the order in which they have been enumerated in the specification of the recovery block. The outputs of each of the alternatives are validated by an acceptance test before being returned to the calling module. If they fail the test, the module rolls back to the beginning of the recovery block, hence undoing all modifications in the variables made by the alternative. It then executes the next alternative. If none of the alternatives is able to produce an acceptable result, an error is signalled to the procedure that called the recovery block. Recovery blocks can be used to provide both forward and backward error recovery.

Figure 2.3 shows a recovery block with two alternatives\(^2\). In this example, if the outputs of alternative AP fail the acceptance test AT, alternative AQ will be used instead. If it does not produce an acceptable result, the recovery block A will return

\(^2\)Generally, there may be one or more alternatives. The simplest possible recovery block consists of the acceptance test and only one alternative.
A: ensure AT by
else by
else error

AP: begin
program text
end
AQ: begin
program text
end

Figure 2.3: Example of a Recovery Block

an error indication. Recovery blocks can be nested, i.e., the alternatives can contain recovery blocks.

The primary alternative will always be the first one to be invoked by the recovery block. Therefore, it ought to be the one which provides the most suitable result. The others, if necessary, may lead to less accurate, although acceptable, results. If all alternatives produce equally acceptable results, the cheapest to compute must be the primary alternative.

Assuming that faults will manifest themselves only in a very limited set of inputs, most of the time the alternatives will produce correct results. Hence, when an alternative fails, it is not excluded from the recovery block, because in the long run it may be less costly to roll the module back and execute another alternative occasionally than to use a less efficient alternative every time the recovery block is called.

Acceptance tests do not guarantee the correctness of outputs that they have validated. They merely state that the outputs are in accordance with pre-established criteria, such as belonging to a particular set of values, or having their elements arranged in a specific order. Thus, erroneous outputs may sometimes be accepted as valid due to the limitations of the assessment criterion. However, correct results will never be rejected, unless the acceptance test is faulty. The acceptance test must be as simple as possible in order to avoid design faults. When recovery blocks are nested, incidental errors caused by faults in acceptance tests may be recovered in outer recovery blocks. It is impossible to design an acceptance test which acts as an
oracle, because if it were capable of foreseeing the correct output for every possible input, there would be no need for the recovery block in the first place.

An alternative may change the contents of global variables. However, if its output is rejected by the acceptance test, the original value of these variables ought to be restored during the rollback procedure. This can be achieved by means of an algorithm similar to a database transaction called recursive cache. A recursive cache mechanism can be implemented by creating a temporary buffer where the new values of the global variables are held whilst the alternative is active [Ran75]. Whenever a global variable is read, it is done either from its permanent address, if it has not been changed by the alternative or, otherwise, from the buffer. The values stored in the temporary buffer are made permanent only if the alternative is successful, else they are discarded.

In real-time applications, the sequential execution of the alternatives may be inadmissible, due to the time taken to roll back the application to the beginning of the recovery block and to execute the next alternative in line. In order for this approach to be suitable for real-time systems, the recovery block must be able to comply with the time constraints of the application in the worst-case scenario, i.e., when all of the alternatives need to be invoked.

During its execution, the alternative may need to communicate with other components of the system. Since this interaction occurs before the output of the alternative is assessed by the acceptance test, it is possible that it contaminates other components by exporting erroneous values to them. The contamination can spread even further if the recipient exports messages to other processes. Contamination can also occur in the opposite direction. The alternative may utilise values imported from other components. Should these values be incorrect, it is likely that the outputs of the alternative will be rejected by the acceptance test. Moreover, if other alternatives obtain these values from the same source, none of them will be successful for the same reason. Thus, the recovery block mechanism must en-
sure error confinement [LK86], i.e., that erroneous data is not transmitted to other components.

2.4.4 Conversations

Depending on how modules in a fault-tolerant concurrent system based on backward error recovery interact with one another, an error detection in one of them may cause many, or even all the modules to roll back to further behind their last checkpoints, including those which have not interacted directly with the faulty module. This bizarre behaviour is known as the domino effect [RLT78]. Figure 2.4 depicts a configuration where the domino effect occurs. Processes A, B, and C are somewhere ahead of their fourth recoverable checkpoint. If an error is detected in process C, it will roll back to checkpoint number 4. However, as it has communicated with process B after checkpoint 4, B may have been contaminated. Hence, B must roll back to the closest checkpoint before its interaction with C, which is checkpoint number 3. B may have contaminated A, because they have communicated after B has exchanged possibly wrong messages with C. Therefore, A must roll back to its checkpoint number 3. Continuing with this analysis, it can be seen that all processes will eventually roll back to checkpoints number 1.

The domino effect can happen when the interaction of fault-tolerant components in concurrent systems is not co-ordinated properly. Conversations [Ran75] is a technique for structuring communicating processes in order to contain contamination. Conversations establish well-defined boundaries within which communicating components are allowed to interact. Communication outside conversations is forbidden. The components that participate in a conversation are checkpointed when they enter the conversation. If an error is detected in any of the components, all members

---

*A recoverable checkpoint* is a checkpoint where a reliable state of the process is not discarded when a new state is logged at another checkpoint. It allows for processes to roll back to checkpoints prior to the last one.
of the conversation roll back to the beginning of the conversation. Components that are not members of the same conversation cannot contaminate one another, because they do not interact. This ensures that members of a conversation do not need to roll back to checkpoints located before the beginning of the conversation. The domino effect is thus limited to the beginning of the conversation. When the members leave the conversation, they are assessed and checkpointed again in order to prevent them from rolling back into the conversation due an error which occurs after they have left the conversation. If a member is found to be in error after it has left the conversation, none of the other members are affected.

Components can join a conversation at different times, provided that they do not communicate before having done so. However, they must leave the conversation at the same time in order to ensure that none of the members are required to roll back into the conversation after having left it. If this rule is not enforced and an error is detected when one of the members is leaving the conversation, members which have already left the conversation will need to roll back to the beginning of the conversation. However, rolling these components back to the beginning of the original conversation would affect the members of conversations that they might have joined after they have left the original one.

Thus, the two rules that must be observed in a conversation stand as follows:
Figure 2.5: Processes Transactions via Conversations

1. Processes must join the conversation before they interact.

2. All processes must leave the conversation at the same time.

Figure 2.5 illustrates two conversations: one between modules A and B, and another between components B and C. Modules B and C join the conversation at different moments (a and b), but communication between them commences only after C has reached point b (rule 1). In order to comply with rule 2, they leave the conversation at the same time (c and d). The same applies to the conversation between modules A and B. Module A is allowed to join its conversation with component B at point e, before the conversation between B and C terminates. However, communication between A and B commences only after B has reached point f.

As a backward error recovery technique, conversations suffer from the same real-time limitations that are present in the canonical implementation of recovery blocks. Another drawback of this software fault-tolerance technique is to roll back all the members of a conversation when an error is detected in one of them, since, theoretically, only the faulty member and those which have been contaminated ought to roll back. The impossibility of determining where the fault that caused the error is located is due to the fact that, notwithstanding the confinement of error propagation,
contamination does occur within the conversation.

2.4.5 N-Version Programming

N-version programming [Avi85] is another software fault-tolerance technique based on design diversity. In this approach, $n$ diversely designed versions of the same module are executed concurrently and their individual outputs are compared in order to decide upon a single result. If no consensus is reached, an error is indicated.

N-version programming is the software counterpart of the hardware technique *Triple Modular Redundancy (TMR)* [RLT78]. Figure 2.6 depicts a TMR fault-tolerant hardware architecture where three parallel channels are utilised. Modules 1, 2, and 3 are diversely designed components which comply with the same specification. Each of the versions receives the same set of inputs and provides its own outputs. In order to ensure that the three channels will contain the same information after the three modules have issued their outputs, each channel has a *voter* at which the outputs of the three versions are compared to one another in order to produce a reliable result.

The efficiency of the N-version scheme is directly related to the independence
Chapter 2. Review on Software Fault-Tolerance

of the versions. Voting on outputs of versions with absolutely independent failure modes will either produce a correct result or an error indication, because two or more of the versions will never fail under the same inputs. However, statistical independence may not be achieved and the versions may show coincident failure modes [KLS85, KL90]. Even then, N-version programming has proved to be an effective technique for improving software reliability [AL88].

Erroneous versions should not be excluded, because they may contribute towards achieving a reliable output when other versions fail. If their behaviour is deterministic, there is no problem in keeping them even when the same inputs are used, because, as in previous occurrences, their incorrect outputs will be masked by the remaining versions.

Voting algorithms are normally based on majority, average, and weight. Majority voting can be achieved by using a bit-by-bit comparison. However, more flexible approaches have to be applied when the same information can be represented in distinct formats, e.g., different precision in the notation of numeric values or spelling variations. Averaged voting is ideal for situations where there is no single correct output, but rather an interval within which any value is acceptable. The unequal degree of confidence that the designer may have on each of the versions can be taken into account in the voting procedure by assigning different weights to their outputs. Although fault-tolerant frameworks may offer generic voting algorithms, such as bit-by-bit comparison, it is essential that they support user designed voting procedures [Lob89] for situations where the voting algorithm needs to know the semantics of the versions.

Voting can be either explicit, by indicating in the code the values that ought to be compared and the moment in which that must happen, or implicit, when the values to be compared as well as the moment are decided at execution time. The DEDIX [AGK+85, TAK86] system, for instance, uses explicit voting, whereas systems such as DELTA-4 [PD85], and CRONIC [Ani89] have adopted transparent
voting strategies.

Voters are single points of failure. In distributed systems, they are to be replicated over different machines in order to survive hardware crashes. However, if the designer is sufficiently confident that they are much more reliable than the versions that they are assessing, which is usually the case, the replicas do not need to be diversely designed.

It is important to correct the state of erroneous versions before their next interaction with the environment. This can be achieved by a technique called community error recovery [TA87], in which the outcome of the voting is returned to the versions. The drawback of this approach is to feedback incorrect values that resulted from similar errors in the majority of the versions. However, assuming that the faulty versions are likely to fail again due to the effect of their previous outputs, one concludes that the final decision of the voter would be the same with or without community error recovery.

DEDIX and Circus are two well-known systems which utilise N-version programming to attain tolerance to residual software design faults. In spite of sharing the principles of N-version programming, the two systems follow different approaches in a number of issues, such as the determination of the values to be voted and time at which the voting shall occur.

DEDIX

DEDIX [AGK+85] is a distributed testbed which has been developed with the purpose of investigating and evaluating various fault-tolerance techniques. DEDIX provides application modules with an intermediate layer located between the application layer and the operating system. Each node of the network has its local copy of DEDIX. The internal states of the diversely designed versions are assessed at cross-checkpoints. The versions call Cross-Check-functions to report to DEDIX
the values that are to be assessed. These values are passed to DEDIX in *cc-vectors*, which hide the internal representations of the values to be assessed from the assessment procedures. Such representations ought to be dissimilar in order to enforce design diversity. However, the need to make explicit calls to Cross-Check-functions, naming the values that are to be assessed, restricts the use of the dissimilar versions to the DEDIX system. Configuring a version in a machine where DEDIX is not executing requires changes in its code.

In order to be able to manage simultaneous Cross-Check-function calls from several versions belonging to different application modules, the design of DEDIX needs to be complex enough to make it possible that some residual faults remain after extensive testing. This possibility may demand that DEDIX itself be implemented by diversely designed versions at a different nodes of the network. The number of versions, however, does not need to be equal to the number of nodes. Other models that provide fault-tolerance by intercepting operating system kernel calls [RSS93] show the same problem.

DEDIX detects errors in the time domain by means of time-out clauses. The time-out countdown can start immediately after the request is sent to the versions (absolute time counting), or after the majority of versions have produced their outputs (majority time counting). Majority time counting prevents communication bursts and excessive node workload, which delay the response of the versions, from causing the indication of an error in the time domain. However, should most of the versions fail completely to reply due to a malfunction, a deadlock situation will arise. This situation is avoided by employing absolute time counting. This approach, however, suffers from the very drawback that is avoided by majority time counting.
Circus

Circus [Coo86] is a system for implementing replicated procedure calls, which are a form of remote procedure call where the remote procedure is realised as a set of replicas. The replicas can be identical, in which case only fail-stop hardware failures [Sch83] are tolerated, or dissimilar implementations of the same specification in order to tolerate design faults.

The communication interface provided by Circus is the same both for single and for multiple versions, thereby hiding replication from the caller. Remote procedure call is an elegant way of providing inter-process communication, but there are situations in which this paradigm cannot be applied without abandoning its synchronous character; e.g., when the caller wishes to resume its computation before the reply is received, leaving it to be read at a later stage.

In a client-server configuration with $m$ diversely designed versions of the client and $n$ dissimilar versions of the server, each server receives $m$ request messages, one from each of the $m$ client versions. The $m$ versions of the request message are, then, reduced to a single message by a collator before being imported by the server version. Reply messages follow an analogous procedure. The drawback of configuring the collator at the destination is that there must be a collator attached to each recipient module, even if it is implemented as a single version. Moreover, should it be possible for an $N$-version server to receive requests from more than one client, each of which may or may not be multi-version, the $n$ collators must communicate in order to guarantee that all the versions of the server import the requests from the different clients in the same order.

Messages may arrive at the consumer in a different order from that in which they have been issued by producers. The sequence in which the messages are received by the diversely designed versions determine the sequence of state transitions made by the versions. Thus, if the order in which messages are imported is not the same
Figure 2.7: An Example of Two Groups Sending Messages to a Third

for all the versions, they may move to dissimilar states and, thus, start exporting messages which, although correct, cannot be compared.

Figure 2.7 illustrates a configuration where a consumer group C, formed by three members (C₁, C₂, and C₃), receives messages from two different producer groups (A, and B), each composed of two versions (A₁ and A₂, and B₁ and B₂, respectively). In some network topologies, it is reasonable to suppose that, if all versions of groups A and B issue a message at approximately the same time, C members will import it in a random order, such as:

\[
\begin{align*}
C₁ & : m_{A₁}, m_{A₂}, m_{B₁}, m_{B₂} \\
C₂ & : m_{A₁}, m_{B₁}, m_{A₂}, m_{B₂} \\
C₃ & : m_{B₁}, m_{B₂}, m_{A₁}, m_{A₂}
\end{align*}
\]

Although all sequences are different, versions C₁ and C₂ will certainly go to the same state, because they will vote the messages from A first. However, version C₃ will vote the messages from B before those from A. Hence, C₃ may go to a different state, thereby losing its synchronisation with C₁ and C₂.
Chapter 2. Review on Software Fault-Tolerance

2.5 Summary

In this chapter, a collection of software fault-tolerance principles and techniques was presented and discussed. Different approaches towards achieving tolerance to software faults and for providing continuous execution of software modules despite hardware failures have been highlighted in the chapter. A taxonomy of these techniques, illustrating how they relate to each other, is shown in figure 2.8. It is important to notice that the techniques described in this chapter are not mutually exclusive. In fact, as it will be shown throughout this dissertation, they complement one another.
Chapter 3

Software Components

Designing fault-free software is virtually impossible in complex systems. Fault-avoidance and fault-tolerance techniques can be employed to improve the software reliability, but it must not be expected that they will guarantee that no fault will ever lead the system to fail. Whilst it is acknowledged that it would be desirable if there could be some metrics for software reliability, the abstract nature of software limits the use of methods such as statistical modelling for its prediction.

This chapter presents a black box classification of software components and comments on the difficulty of quantifying software reliability. In addition, it identifies the factors that might be responsible for the occurrence of faults in software systems. Finally, it advocates the use of diversity in the various phases of the software development process and the selective and transparent introduction of fault-tolerance to software.

3.1 Black Box Classification of Software Modules

One of the most common ways of viewing a software module is as a black box that maps a set of inputs $I$ into a set of outputs $O$ according to a function $f$, as depicted
Figure 3.1: Black Box Model of a Software Module

in figure 3.1. When the output \( o \in O \) is a function solely of the input \( i \in I \), indicating that the module does not retain state, it is said the module is \textit{history-independent}. Conversely, when the module retains state, its output is a function not only of the inputs but also of the state of the module \( s \in S \). Since the past history of the computation of the module affects its outputs, modules that retain state are also referred to as \textit{history-dependent} modules.

In brief, a software module can be classified, as far as the past history of its computation is concerned, as:

- \textbf{history-independent}: \( o = f(i) \)
- \textbf{history-dependent}: \( o = f(i, s) \)

As far as predictability is concerned, software modules can be divided into \textit{deterministic} and \textit{non-deterministic}. Output \( o \in O \) in the former will always be the same given the same input \( i \in I \) and, for history-dependent modules, the same state \( s \in S \). In the latter, random variables make it impossible to determine output \( o \in O \) solely based on the input to the module and on its current state. In addition, for history-dependent modules, input \( i \in I \) occurring when the module is at state \( s \) \textit{in} \( S \) will take the module to a particular next state \( s' \) \textit{in} \( S \) only if the module is deterministic. Thus, if \( R \) is the set of random variables that may influence the behaviour of a software module, such a module is classified as:

- \textbf{deterministic}, if \( o = f(i) \) or \( o = f(i, s) \), and
- \textbf{non-deterministic}, if \( o = f(i, r) \) or \( o = f(i, s, r) \), where \( r \in R \).
A software module may perform a function which does not require any input or which does not produce any output. In the former case, the module is classified as a producer (figure 3.2.a), whilst in the latter it is referred to as a consumer (figure 3.2.b). Software modules may provide services to other components by receiving inputs from them and sending to them the outputs of the computation. These modules are called servers (figure 3.2.d) and the components that asked for the service are named clients (figure 3.2.c). Filters (figure 3.2.e) are software modules that receive inputs from one module and transmit the output of their computation to a different module.

In summary, a software module can be classified, as far as its inputs and outputs are concerned, as:

- **producer**: $I = \emptyset$ and $O \neq \emptyset$.
- **consumer**: $I \neq \emptyset$ and $O = \emptyset$.
- **client**: $I \neq \emptyset$ and $O \neq \emptyset$, and output $o \in O$, sent to module $S$, causes input $i \in I$, sent by module $S$. 

Figure 3.2: Black Box Model Classification as to its Inputs and Outputs
Chapter 3. Software Components

- **server**: \( I \neq \emptyset \) and \( O \neq \emptyset \), and input \( i \in I \), sent by module C, causes output \( o \in O \), sent to module C.

- **filter**: \( I \neq \emptyset \) and \( O \neq \emptyset \), and input \( i \in I \), sent by module A, causes output \( o \in O \), sent to module B.

The role played by a software component does not need to be the same throughout its computation. A file server, for instance, performs the role of a server in requests received from an application module, but it takes a client position in its interaction with a disk server. Moreover, the classification of a software module in terms of its inputs and outputs is not absolute. It depends on the cause-effect relationship between the input and the output. Thus, server and filter components are classified as such only if the output is issued as a direct result of the reception of the input. An analogous criterion can be used to classify a software module as a client. If the influence of the input on the output is not direct, components that import and export data at different points of their code can be classified as software modules that play alternate roles as consumers and producers during their computation.

### 3.2 Estimation of Software Reliability

Several models have been proposed to estimate software reliability [AGCL86, Tra90, Puc90, BJ90]. However, the accuracy of these models is not as great as that of models used to predict hardware reliability.

The data used to estimate hardware reliability derive from observation of component degradation, which is a physical property. Thus, it is possible to assign reliability figures, such as MTBF (mean time between failures), and MTTF (mean time to fail) to hardware devices based on the statistical analysis of data collected from past experience with other circuits with the same design and assembled according to the same manufacturing technique.
Conversely, as software does not degrade with time, software faults are already present when the system becomes operational, although the moment in which a fault will cause an error is unknown. The software failure process is random [Lit79]. Errors in a program occur when the program, at particular states, is subjected to a particular set of inputs. The parameters of probabilistic models based on this assumption are determined from error logs. However, the utilisation of error logs to forecast the future behaviour of the software has limited value, because the program changes whenever a fault is removed. Alternatively, since the number of conceivable inputs that can be tested in practice is too small to ensure a low probability of failure, one can use \( n \) random tests to calculate the probability that a program with reliability worse than a particular value would pass the \( n \) tests [PvSK90]. Unfortunately, under plausible mathematical assumptions, the chance that a program will execute without an error for the same time that it has executed before is around fifty per cent [LS92]. In addition, the assumption of a constant failure rate over time that is made in those tests does not hold due to the existence of contiguous failure regions in the input space [Bis93].

The lack of confidence on precise numeric figures for software reliability is acknowledged by regulation agencies which have opted for ascertaining software reliability qualitatively by assessing the development process, rather than measuring it quantitatively. In civil aviation, for instance, the American Federal Aviation Administration (FAA) requires the probability of a hardware failure to be less than \( 10^{-9} \) per hour of flight [Fed]. However, this requirement is not applied to software because the FAA regards it as infeasible to assess the number or kinds of residual software faults. The Radio Technical Commission for Aeronautics equally considers a disciplined approach to software as more critical than quantitative methods [Rad].

Notwithstanding the difficulties of quantifying the reliability of a program as an absolute measure, mathematical models and practical experiments can be used to compare the reliability of a software system with other functionally equivalent programs. Theoretical [EL85] and empirical [AL88, ABHM88] studies have indicated
the potential gains in software reliability from the use of fault-tolerance techniques based on design diversity, as well as their limitations [KL86].

3.3 Software Development Process

The process of software development can be described as a progression that starts from the user's requirements and passes through a series of stages, namely specification, and design until the final code is ready to be delivered to the user. The software design process is not an unidirectional activity. It is usually necessary to return to previous stages in order to review decisions that have proved to be either incorrect or inadequate at later stages. Even after the software has become operational, corrections and upgrades may become necessary. Therefore, maintenance is also responsible for changes in the design and implementation of software systems.

Faults may be introduced in all phases of the software life-cycle. The development of a software system starts from the user's requirements, which tend to be a loose collection of functions that the user wants the application system to perform. The user's requirements are commonly communicated orally or in draft written form. Thus, they lack a number of details which are inadvertently omitted for being common sense in the community in the area of the application. Some of these details may lead to design faults, because they refer to information that is not obvious or known to designers who are not familiar with the topic.

The next step in the software development process is to produce a specification on which to base the design of the system. The specification gives a precise indication of the functional behaviour of the software without determining the means by which it will be accomplished. The specification is a translation of the user's requirements into a formal document. The specification may be produced exclusively by the user,

\[1\]The term user in this context refers to the person or people who benefit from the utilisation of the system and not necessarily to whom operates it.
although computer experts are usually involved in the process. The specification may be expressed in many formats, such as plain language, functional diagrams, and formal specification languages. Specifications may be incomplete or incorrect in parts. These may be attributed either to the incapacity of the user to foresee all possible situations in which the system will be utilised, or to the lack of essential information that the user assumed to be known by the designer. In addition, when the specification is written by the designer, or a third party, faults in the specification may arise from misunderstandings of the user's requirements.

The software design, which must comply with the specification, can be divided into two levels: system design and module design. In a top-down approach, the system design stage consists of defining the subsystems and their interactions. The same methodology can be applied to the subsystems until primitive components are reached. Incorrect definition of subsystem interfaces, interconnection, and synchronisation will result in faults in the system design. Potential deadlock situations that are not foreseen and avoided at design time may also cause errors in the software during its operational phase. Primitive components are realised as sequential programs. Faults in the design of primitive components are caused, for example, by the utilisation of inappropriate algorithms or data structures, and inaccurate specification of subroutine interfaces.

The final stage in the software development process is the implementation (or coding). Faults in this stage reflect, in many cases, the inability of the programmer to realise the algorithm defined by the designer and accidental typing mistakes that are not detected at compilation, because they comply with the syntax of the programming language. Other sources of faults in the executable code are compilers and libraries. Some of the faults in compilers and libraries are documented, but it is possible that the programmer is not aware of them. More serious than these are faults which have not been detected either by the developers of compilers and libraries or by their users.
Chapter 3. Software Components

Maintenance is an essential part of the software life cycle, because it is responsible for removing faults whose manifestations have been observed only when the software was already in operation. Faults are fixed by changing the implementation of the software and, sometimes, its design and even its specification. Therefore, it is possible that new unintentional faults be introduced. Although most of these new faults are eventually removed, it is not uncommon to see the reliability of the system declining when a fault is removed. This possibility is acknowledged in some reliability growth models [AGCL86].

3.4 Diversity

Fault-tolerance techniques (chapter 2) based on diversity have been used to improve software reliability. They rely on the principle that dissimilar programs that realise the same user's specification are unlikely to fail under the same inputs. Since faults may be introduced in any phase of the software design process, diversity may need to be applied in all of them, namely:

- specification
- design
- implementation

The most obvious phase of the software development process in which diversity must be used is the implementation. However, if the task of writing the dissimilar programs is given to the same person, it is likely that he/she will, consciously or not, repeat the same structures in all replicas, thereby producing versions with a number of sections with no diversity at all. Thus, the dissimilar implementations must be written by independent teams of programmers who, ideally, should not share common educational, professional, and cultural backgrounds.
Errors caused by faults in compilers and libraries are tolerated by using different compilers and libraries for each version. Programming languages determine, to a certain extent, the implementation of a particular algorithm. Languages that support recursion, for example, allow for recursive algorithms to be implemented either in a recursive fashion or as a loop structure, whereas languages in which recursion is not possible leave the programmer with no option but to implement algorithms that are recursive by nature iteratively. Thus, writing the dissimilar versions in different programming languages would be an advantage.

Utilising different programming languages, compilers, and libraries to enhance diversity at the implementation level may lead to assembling problems when the system is configured, if the files produced by the different compilers are to be linked at compilation time. Thus, in order to overcome this potential limitation, the dissimilar implementations ought to be linked at execution time. Therefore, the programming paradigm in which fault-tolerant software will be written must be capable of supporting dynamic configuration. The basic features of programming paradigms for building fault-tolerant software are discussed in section 5.1.

Diversity must also be used at the design level, otherwise conceptual faults, or limited application of the algorithm may prevent the dissimilar implementations from fulfilling the user's requirements, even if they realise the algorithm with absolute correctness. It is clear, however, that the diverse designs ought to satisfy the user's requirements. The task of designing each version should be assigned to separate teams with diverse backgrounds, for the same reasons as teams of programmers should. Needless to say that the teams must not communicate during the design process.

As mentioned in section 3.3, faults may arise when the user's requirements are translated into a specification. Thus, various specifications of the user's requirement may be produced in order to avoid that imperfections in one of them result in common faults in all the versions. As in the design and implementation stages, the
dissimilar specifications ought to be written by different teams, possibly in different formats to enhance diversity. Ideally, the diverse specifications should be compared mechanically, if possible, or manually, otherwise, with one another before the design of the versions start so that specification faults are eliminated in the early stages of the software development process. Once the specifications have been found to be equivalent, any of them could be used by the various design teams as the basis for their work. However, unless there is a specification which is overwhelmingly superior to the others, each design team should adopt a different specification in order to prevent a common fault which has not been detected in the comparison procedure from causing the diversely designed versions to fail under the same circumstances.

Whilst it suffices for fault-tolerance mechanisms at execution time to assess the diversely designed versions at specific points of their computations for a particular input at a time, it is necessary to consider all possible inputs for each function of the system, if all dissimilar versions are to be designed from a single specification.

The use of diversity in the process of specifying, designing, and implementing software systems can be seen in the diagram of figure 3.3. Since implementation can be seen as low level design [Ran75], the design and implementation phases in the software development process are very much interrelated. Thus, each dissimilar implementation usually derives from a different design. Conversely, the number of different specifications has been usually smaller than the number of diverse designs [Avi85, KL86].

### 3.5 Selective and Transparent Fault-Tolerance

Notwithstanding its importance as a means to increase the reliability of software systems, fault-tolerance is time consuming and costly. Therefore, the introduction of software fault-tolerance must be selective, i.e., only components which perform critical functions should be made tolerant to faults. Although the system would operate in a degraded mode when a non-critical component failed, its behaviour
The concept of critical function varies greatly according to the application. In addition, the boundary between critical and non-critical functions is not always clear. For example, there is no doubt that in fly-by-wire control systems, such as those of the A320/A330/A340 AIRBUS family [BT93], the modules that supervise and command the flight control surfaces are critical, whilst fault-tolerant software would hardly be developed to control the command pads used by the passengers to select audio and video channels for entertainment during the flight. However, in an air traffic control system (see appendix D), there could be disagreement as to whether the radar control system is critical or not. One could argue that an error which resulted in the incorrect display of radar tracks could confuse the air traffic controller to the point where collisions could occur, to which others would reply that this situation would hardly occur because the controller must regularly confirm the aircraft position via radio contact with the pilot.

Producing software systems based on reusable components reduces development
cost and time and, more important, has the advantage of utilising components which have been observed in other systems. Thus, in addition to selective, fault-tolerance should be as transparent as possible. This implies that other system components should not be required to have special interfaces to communicate with fault-tolerant modules, as this would restrict their usage to fault-tolerant systems.

As to the diversely designed versions, which are functional modules, the code for providing fault-tolerance to them should not be intermixed with their functional code. In addition, if the dissimilar versions are to be used without modifications both as versions of a software fault-tolerant module and as the single component in the system to perform a particular function, they must have the same external interface in both configurations. Thus, as far as functionality is concerned, the fault-tolerant module as a whole must be interchangeable with any of the versions individually.

History-dependent modules, however, cannot always be used as versions of fault-tolerant modules without alteration. A faulty version may move to an erroneous state when a fault manifests itself. If the version does not recover from the error by moving to the correct state, it may never comply with its specification again. Since errors in other versions may occur, possibly caused by different faults, it may eventually become impossible to provide fault-tolerance. Hence, erroneous versions which retain state must recover before they resume their computation.

Analogously, fault-tolerance cannot be transparent to non-deterministic modules, because the random variables in their computation may take the versions to non-comparable states. The versions will then lose synchronisation. Fault-tolerance can be transparently attached to non-deterministic history-independent modules, provided that the algorithm used to assess their outputs be capable of reducing the random outputs of each of the versions to a single output to be delivered to the rest of the system.
3.6 Summary

This chapter concentrated on software components, their classification, and development process. Three classifications of software modules were presented. As far as the influence of state on their current computation is concerned, software components can be either *history-dependent* or *history-independent*. In terms of the predictability of the outputs of the modules, software components can be divided into *deterministic* and *non-deterministic* modules. As to the origin of the inputs received by them and destination of the outputs that they issue, software components are classified as *producers, consumers, clients, servers*, or *filters*. A software component may play distinct roles at different points in its computation.

The logical character of software makes it difficult to measure reliability as a numeric value, such as MTTF, for instance. Hence the importance of the techniques employed in the development of critical software systems, amongst which fault-tolerance mechanisms. As software does not degrade with time, residual faults can only be tolerated if the redundant versions are designed and implemented according to dissimilar strategies that comply with a unique requirement from the user. In many cases, it may also be justifiable to apply diversity in the specification phase of the software development process.

Fault-tolerance must be selective and transparent in order to allow software modules that do not require a high degree of reliability to be configured alongside fault-tolerant components without changes in their design. Analogously, one should attempt to design functional components so that they could be configured either as diversely designed versions of a fault-tolerant component, or each of them as the only module to perform a particular function in a non-fault-tolerant application.
Chapter 4

Modelling Fault-Tolerant Software Components

Fault-tolerance mechanisms have been effectively used to improve the degree of availability and reliability of software systems [Dav84, FL84, TBH87, Hag88]. There are several well-known techniques for tolerating software malfunction. The key techniques were presented in chapter 2. No fault-tolerance technique, however, is suitable for all types of faults. The choice of the most appropriate one varies according to the type of fault and to the damage caused to the application by the manifestation of faults.

The designer may need to choose a different software fault-tolerance technique for each part of the system. In addition, the limitations of these techniques can, in many cases, be reduced by combining or augmenting them. Thus, the process of designing and implementing software fault-tolerant systems is facilitated by bringing the principles of the various fault-tolerance mechanisms together into the same framework. Furthermore, such a framework disciplines the development of fault-tolerant software, thereby reducing the risk of introducing faults in this process. This chapter aims to model this framework, investigating the necessary components and their interactions.
Chapter 4. Modelling Fault-Tolerant Software Components

4.1 Framework Outline

Fault-tolerant software systems are intrinsically more complex than their single-version counterpart. However, if the system is structured in hierarchical levels of abstraction [PT91], its overall complexity is decomposed and the complexity of each abstraction is hidden from the others. If a component is replaced by another one with the same interface, the degree of complexity of the abstraction which contains it remains unchanged, even if the new component is internally more complex than the original one. Thus, components which have to tolerate software faults must be implemented as composite components that encapsulate the diversely designed versions.

The outputs produced by the versions must be assessed before they leave the composite component, in order for the composite component to be logically interchangeable with any of the versions (section 3.5). Equally important is to avoid common errors in the computation of the dissimilar versions which are caused by inputs with invalid syntax or incorrect semantics. Therefore, assessment components must be present in fault-tolerant modules.

There are situations where no acceptable output is obtained from the diversely designed versions. Then, the fault-tolerant module ought to produce a default output or, at least, report the failure to external modules. In addition, should a version that retains state fail to produce a correct output, it ought to be capable of recovering from the error. Thus, software fault-tolerant modules must include a third type of component, which caters for error handling.

In order to maximise reusability, the design of the components of a software fault-tolerance framework must depend as little as possible on the design of its other

---

1 Single-version software is the term used in this dissertation to refer to non-fault-tolerant software, in which each function is implemented by a single software module.
components. Thus, the dissimilar versions must not be required to be aware of the fact that their outputs will be assessed. Dissimilar applications may need different assessment and error handling components. In addition, the order in which the versions execute and the sequence in which their outputs are assessed may vary according to the application. Hence, the versions, assessment components, and error handling modules are, then, to interact under the control of a *co-ordination component*.

Figure 4.1 outlines the generic framework of a software fault-tolerant component. The blocks on the right represent the diversely designed versions. The top component represents the assessment algorithms. The bottom box symbolises the error recovery components. The element that co-ordinates the interaction amongst these three subsystems is depicted by the central rectangle. In order to demonstrate the utility and generality of this framework, it is necessary to illustrate the possible forms of each of these subsystems. As a result, the next three sections will address:

- assessment
- error handling
- control

### 4.2 Assessment

The only way of being absolutely sure that an output produced by a software module is correct is to use an oracle which is capable of foreseeing the theoretical value. During the implementation and testing phases, the designer acts as the oracle. In the operational phase, however, it is impossible to use an oracle. If a program were capable of producing the correct output for all possible inputs with absolute certainty, there would be no need for fault-tolerance. In the absence of an oracle, error checking becomes a critical activity of fault-tolerance mechanisms.
4.2.1 Output Assessment Strategies

The correctness of the outputs of a software module can be assessed by conformity tests, which check whether the outputs conform with a certain criterion, e.g., if they are within a certain range, whether they belong to a particular set, whether they are sorted in a particular order. Recovery block acceptance tests (section 2.4.3) are based on this approach. The drawback of this method is that it is possible to envisage incorrect outputs which comply with the conformity criterion. For example, if the assessment criterion requires that the outputs produced by the versions belong to a particular set of values, any of these values will always be regarded as acceptable, even when it is not the correct one.

Another means of assessing component outputs is by voting. If a consensus is reached, the user can have more confidence, although not absolute certainty, in this value than if only one value were available. Should the voting algorithm reach no consensus, nothing can be said about the correctness of any of the values. A variant
of this form is to weight or average individual outputs.

Conformity tests are highly application dependent. Although some of them can be flexible enough to deal with a whole class of algorithms, such as sorting algorithms, it is not possible to design a universal conformity test. Unlike conformity tests, some voting strategies are generic. For example, a bit-by-bit comparison algorithm can be used to vote the outputs of any type of version, since it requires them to match completely. However, some voting algorithms have their use restricted to particular types of versions. Cosmetic voting [AGK+85], for instance, allows for character strings to be slightly different due to words misspellings or character substitution. It cannot be implemented unless the rules for acceptable misspellings used by the versions are known.

Voting is more precise than the conformity test, because the pattern that the output of each version ought to match is determined by the outputs of the remaining versions. However, voters do not ascertain the plausibility of the outcome of the collation, whereas conformity tests do. For example, an out-of-range value produced by most of the versions would be regarded as reliable by a majority voter, whereas a conformity test would reject it.

Naturally, both methods can be combined. One possible approach is to validate each individual output first, by submitting them to a conformity test, and to vote them afterwards. Although the outputs which have been rejected by the conformity test do not need to be submitted to the voting procedure, an indication of such errors must be made to the voter so that it can decide whether the quorum to the voting has been achieved. Therefore, this approach is only suitable for fault-tolerant systems in which the voting does not require that the majority of the versions agree on the same output. For example, in a fly-by-wire aircraft control system, in which a value that may be incorrect is better than none at all, it is preferable to accept a plausible value produced by one version out of three when the other two candidates are definitely incorrect (e.g., out-of-range values). Otherwise, it is more efficient to
vote the outputs and then to submit the outcome of the voting to the conformity test. By coupling the two assessment strategies one can exploit the greater precision provided by the voting technique, whilst avoiding that values which do not belong to the universe of acceptable outputs be validated.

Interface Differences

Conformity tests and voters differ not only on their principles, but also on their interfaces. Hence, the control subsystem (figure 4.1) must provide different interfaces to communicate with them. The existence of one must not preclude the configuration of the other, since there are applications that benefit from a combination of both assessment strategies.

The number of versions in the fault-tolerant module does not need to be passed to the conformity test. Conversely, this information is usually important for voters, since they usually need to inspect the majority of their outputs before agreeing on an acceptable value. In addition, in some voting strategies, such as weighted voting, the identification of the version originating the output is crucial. Another example where the identification of the originating component is crucial arises when there is an order of preference amongst them, but no suitable criterion on which a conformity test can be based. In such cases, a regular voting algorithm can be used to determine a pattern that an acceptable output must closely match. Once the pattern has been established, the voter can act as a conformity test and select from each matching output the one which has been produced by the highest priority version. This technique can be seen as an adaptable conformity test, as the conformity criterion is dynamically set during execution time, rather than during the programming phase.

The outcome of a conformity test is the indication of whether or not the data assessed by it comply with a chosen criterion. Conformity tests do not suggest appropriate corrections. The output of a voter, on the other hand, cannot be a
simple "yes-or-no" indication. It has to show the acceptable data value. In bit-by-bit voting, for instance, the voter must report which value has been chosen. The voting outcome is even more important in voting algorithms that generate a new data set from the individual candidates, such as in averaged and weighted voting.

### 4.2.2 Input Validation

Equally important to checking the outputs issued by the versions is to assess the inputs received by them. The assessment of the incoming messages becomes even more crucial in history-dependent modules in cases where no reply from the versions is expected. In these cases, an incorrect computation due to an invalid input will not be detected immediately. Although it may be of no importance at the time the error occurs, since it will have no external side effect, such a malfunction may eventually be responsible for any future misbehaviour of the module when it issues messages or actuates peripheral devices that depend on its previous computation. Moreover, as the same input is broadcast to all the versions, a bad input may invalidate the voting process as an effective means of detecting erroneous outputs.

It is reasonable to expect that well designed versions will have internal input validation procedures, especially if they are also to be used outside a software fault-tolerance framework. However, if the designer of the fault-tolerant module judges that the versions have inadequate input validation procedures, he may wish to utilise more appropriate algorithms. Thus, in addition to a voter and a conformity test, the software fault-tolerance framework depicted in figure 4.1 must provide for an input validation module. Input validation modules are, in fact, conformity tests that assess inputs, rather than outputs. The similarity between the algorithms of the two assessment modules is due to fact that both assess a single data set at a time and are expected to communicate their decision of whether the data set has been found acceptable or not in a boolean format.
4.2.3 Structuring Principles

The framework outlined in figure 4.1 indicates that the assessment of the messages exchanged between the versions and external modules is not done by the versions themselves, but rather by independent modules. This is so in order for modules that have been designed to be configured in single-version systems to be used as versions without requiring them to be customised to fault-tolerant applications. Altering these modules to assess their own messages might lead to faults that do not exist in their original implementations. Their complexity would increase not only because new functions would be added to them, but also because they would probably have to interact with the remaining versions in order to agree upon an acceptable output.

A comprehensive framework for developing fault-tolerant software modules must cater for three different assessment components, namely:

- input validation module
- conformity test
- voter

Thus, the generic internal configuration of the assessment subsystem of figure 4.1 can be represented by the diagram of figure 4.2. However, in order for this framework to be flexible enough to accommodate application modules with different needs, the configuration of these components must be optional. For example, producer fault-tolerant modules (section 6.1), which only export data, do not need the input validation module. In addition, it is not always necessary or convenient to combine the voter with the conformity test to assess the outputs of the versions.

Assessment modules may contain design faults too. Hence, it is important to minimise their complexity so that they can either be proved correct or, at least, be much more reliable than the module that they are to assess, otherwise they may
reduce the reliability of the software fault-tolerant module. Should it be impossible to design and implement conformity tests or input validation modules that are much more reliable than the versions, design diversity must be used. The "unreliable" diversely designed assessment components can be structured as versions of a fault-tolerant component which is based on the same framework studied in this chapter. Since the output of each of them is a boolean value, they could be assessed by a trivial majority voter, as illustrated in figure 4.3. The same approach could be used to provide fault tolerance to "unreliable" deterministic voters, such as those based on a bit-by-bit majority criterion or on weighted voting, provided that all the voters assigned the same weight to the same versions.

Some assessment criteria, such as majority voting based on a bit-by-bit comparison of the individual results, or compliance with an alphabetical ordering, can be applied to versions with very different functionality. Thus, as the versions, the assessment components should be as much as possible designed to be reusable.

### 4.3 Error Handling

Design diversity reduces the risk of a fault leading to a failure, but it does not eliminate the possibility completely. Therefore, should an error occur, the fault-
tolerant module must be able to recover from it or, at least, to prevent it from causing a catastrophic failure. In addition, if the fault-tolerant component acts as a server, irrecoverable errors must be reported to the external module which made the request. Error recovery components are invoked when the input validation module rejects a data set imported from external components or when the voter and/or the conformity test fail to agree on a reliable output.

### 4.3.1 Compensation Modules

Ideally, versions must not take any external action before their computation is assessed. However, components which have not originally been designed to be used in fault-tolerant configurations do not expect to receive feedbacks from their outputs. They will produce the external actions that they are scheduled to, regardless of whether there has been any error in their computation or not. Thus, in order to allow components that have not been designed to delay external actions until they receive a feedback from their outputs to be configured as versions, this requirement
can be relaxed, provided that the effect of an incorrect action on the environment is not serious and that there are simple means of amending it.

The control subsystem of the fault-tolerance framework outlined in section 4.1 is never aware of any external action caused by the versions. Therefore, it has no means by which to compensate for any of them which is incorrect. The control subsystem, however, is capable of sending information to the versions regarding the assessment of their outputs. This information can be used by those versions whose outputs have not been validated to execute a compensation procedure. In addition to the error indication, the version may need the correct output to accomplish the compensation procedure successfully. Hence, the final outcome of the assessment components must be sent to the versions along with the verdict on their outputs. Should no reliable output be agreed upon, the versions must be informed accordingly.

Compensation procedures can be encapsulated in modules that have been designed to be configured as versions. However, it is unreasonable to suppose that modules that do not expect any feedback on their computation, e.g., modules that have not originally been designed to execute within a software fault-tolerance framework, will cater for such procedures. In these cases, compensation must be executed by separate modules, known as compensation modules, on behalf of the versions.

The control subsystem must define a single protocol to communicate the outcome of the assessment to compensation procedures, regardless of whether they are located in the versions or in compensation modules. Moreover, the control subsystem should not even need to know whether it is sending the information to a version or to a compensation module.

Versions can sacrifice transparency and exploit the support for compensation modules to delay external actions until the assessment components have inspected their outputs. Technically, in this case, there is no compensation, but the necessary underlying support is exactly the same.
4.3.2 Internal Error Recovery

History-dependent versions must recover from errors that have occurred before they resume their normal computation (section 3.5). To do so, they must recognise that an error has occurred and, then, move to a correct state. Only when the versions share the same memory space and have coincident internal states can backward error recovery be adopted, because the state will automatically be corrected by a successful version. In diversely designed versions that do not share state, erroneous versions cannot return to a previous, reliable state, because they will lose synchronisation with the correct ones. They must perform an internal forward error recovery to a state that matches the current states of all correct versions. They have to be informed of the exact values of the variables that compose the correct state or receive data that enables them to infer the correct values of the state variables.

In a fault-tolerance framework where the only observable values of the versions are their outputs, there may be versions which are not able to deduce the correct values of the state variables from the feedback of the correct output. This limitation and solution are discussed in section 9.1.2.

Internal error recovery and compensation should not be mutually exclusive. The fault-tolerance framework must allow for the versions and compensation modules to read the same data. Transmitting the same information to different entry points must not, however, require different control components, firstly because it would constrain its reusability and secondly because the same version may not need to perform both activities whenever an error occurs.

4.3.3 Error Handling Modules

Well designed fault-tolerant software components based on diversely designed versions are more reliable than any of the versions individually. Common failure modes are expected to be rare, but they may exist. Thus, the fault-tolerant module must
be prepared to cope with cases where the assessment components do not agree on an acceptable output. Should this situation occur, the fault-tolerant module must report the error to other components of the system or to the environment and, whenever possible, go to a safe state. For example, if the assessment components cannot agree upon the signals that ought to be sent to two trains travelling in opposite directions, instructing them which tracks to take at a junction, the fault-tolerant module must, above all, avoid a crash between the two trains. Following safeguard specifications, the trains could be taken into a safe state by calling a procedure to stop both trains before they reach the junction. In addition, an error message could appear on a supervisor's monitor. Another function that should normally be carried out is to log the error for future analysis. From the error analysis it may be possible to identify the fault or, at least, to recreate the events that resulted in its manifestation.

Error handling is very much application dependent. Thus, the control component of the framework outlined in section 4.1 cannot perform this task without losing its generality. Instead, the framework suggests that components be specially created for this purpose. It is important to notice, however, that these error handling modules are only invoked when the assessment components do not agree on an acceptable output. Individual errors in each of the versions are to be dealt with by the versions themselves and/or by their respective compensation modules, as explained earlier in this section.

Server fault-tolerant modules are usually expected to reply to their clients with an error indication whenever they cannot perform the required service, thereby transferring to the client the responsibility for coping with the error. The error message cannot be automatically created by the control component, because its format is not always the same. The most adequate components to generate them are error handling modules.

More than one error handling module may be needed, because it might be easier
to design and implement separate components to deal with different errors than to develop a single module which must identify the error before choosing the appropriate error handling procedure. There must be always a universal error handling module to deal with unpredictable errors. The error handling module to be called is determined by the assessment components, because they can extract from the individual outputs of the versions the information necessary to classify the error.

An error handling module may be unable to handle the error alone and need to invoke other error handling modules. Therefore, the controller must be capable of recognising a message from an error handling module requesting it to call another error handling module.

4.4 Co-ordination Component

Communication amongst versions, compensation modules, assessment components, and error handling modules is co-ordinated by a module which is referred to as the controller. Designing a new controller for each different application could lead to less reliable modules due to the intrinsic risk of the existence of faults in new code. Thus, the number of different controllers must be as small as possible without, however, leading to generic controllers that cannot be considered as much more reliable than the versions. In order for a controller to be as generic as possible, its algorithm must not depend either on the syntax and semantics of the messages that it exchanges with the versions or on the algorithms of the assessment components and of the error handling modules. Conversely, it needs to know the number of versions and error handling modules and the assessment modules that have been configured.

The outcome of the assessment of the outputs produced by the versions depends not only on the algorithms of the versions and of the assessment components, but also on the order in which the individual outputs are assessed. Since the versions do not communicate with the assessment components directly, it is the responsibility
of the controller to determine the order in which the outputs of the former will be delivered to the latter.

4.4.1 Assessment Order of Outputs

The first output validated by the conformity test is the one to be exported from the fault-tolerant module. Thus, the order in which the outputs of the versions are to be sent to the conformity test must follow a priority criterion. Priority criteria are determined by a number of different factors, such as the precision in which numeric values are expressed, and the adequacy to the problem of the mathematical model used by each version. A method of telling the controller the sequence in which the outputs of the versions must be passed to the assessment components is to enumerate the versions in the same sequence. Lower priority versions will have their outputs assessed only when all higher priority versions have failed the conformity test.

Conformity tests can also be configured to assess versions that produce outputs that are equally acceptable. The priority criterion in these cases is usually response time. As the time taken by each version to issue its output varies according to such factors as the complexity of their algorithms, the processing power of the computer where they are executing, and the delay in the communication between nodes, the order in which the outputs are to be assessed cannot be fixed a priori. Hence, the versions must execute concurrently and their outputs then submitted to the conformity test in a first-come-first-served order.

In a fault-tolerant module based on voting, should the versions have similar degrees of reliability and efficiency, the order in which their outputs are received by the voter is unimportant. Hence, their outputs ought to be sent to the voter in a first-come-first-served sequence in order to minimise the time taken by the voter to reach a consensus. Voters, however, do not preclude the existence of an static order of preference amongst the versions. Such an order is determined by the same sort of criteria that have been considered for conformity tests and, analogously, it is
translated into the order in which the versions are enumerated.

Therefore, the controller must cater for two distinct assessment orders: static, and dynamic. In the former, the outputs are sent to the assessment components in a pre-established order, which is determined by static criteria. In the latter, the outputs are transmitted in a first-come-first-served order to the assessment component.

### 4.4.2 Temporal Sequencing of the Versions

Conformity tests imply some sort of ordering in the execution of the dissimilar versions. Traditionally, versions whose outputs are to be assessed by a conformity test have been configured to execute in sequence. Indeed, since usually the first version succeeds in producing an acceptable output, resources are allocated only when they are necessary. However, should more than one version need to be invoked, the delay to achieve a reliable output may be unacceptable. If this is the case, the versions must execute concurrently so that the waiting time for the arrival of the outputs of lower priority versions is reduced to the minimum.

Voters are assessment components that usually have to wait until the majority of the versions have issued their outputs before agreeing on a reliable output. Thus, the argument in favour of executing the versions in sequence in order to avoid unnecessary resource allocation does not hold in this case. Thus, the versions of fault-tolerant software modules whose outputs are assessed by voters should execute concurrently.

Hence, the framework outlined in figure 4.1 must support two different execution modes for the versions, namely: sequential and concurrent. It is important to notice, however, that the choice for one instead of the other must result from the priority between resource allocation and response time. The criteria for determining the order in which the outputs of the versions are to be assessed are independent from
the ones used to decide the mode in which the versions will execute. However, sequential execution always implies a static assessment order.

4.5 Summary

Software fault-tolerant modules must be constructed under a framework in which the overall complexity is not significantly affected or where complexity can be decomposed. The basic requirements of such a framework were investigated this chapter.

Fault-tolerance is achieved by submitting the equivalent outputs of diversely designed versions to assessment components. The inputs to be received by the versions might also need to be assessed. Therefore, a framework for designing and implementing fault-tolerant software modules has to be capable of supporting conformity tests, voters, and input validation modules. Versions whose outputs are different from the one accepted by the assessment components must be informed of their error by compensation messages.

Should the assessment components be unsuccessful in finding an acceptable output, the fault-tolerant component must go to a safe state and report the error to other components of the system or to the environment. The function of reporting irrecoverable errors should be performed by error handling modules, which are also responsible for generating information to be sent to the versions in order for them to be able to move to a safe state. In order to maximise reusability, the interaction of these components should be co-ordinated by a generic component, called the controller. The controller should not need to know the algorithms of the other components of the fault-tolerance framework, because it would reduce its reusability in other fault-tolerant modules.
Chapter 5

A Software Fault-Tolerance Framework

Fault-tolerant software systems should be organised in modular units so that software fault-tolerance can be selectively and transparently incorporated to modules that perform critical functions. Fault-tolerant components should be structured under a comprehensive framework which supports the principles of well-known software fault-tolerance techniques. Thus, fault-tolerant software modules can be based purely on a single technique or on a combination or augmentation of more than one technique.

The basic requirements and general structure of such a framework were examined in chapter 4. An implementation model for the framework is proposed in this chapter, emphasising the utilisation of reusable components. This is an architectural model upon which tool-kits for building fault-tolerant software components can be developed.

It is also addressed in this chapter the mechanisms for detecting errors in the time domain, and the assessment of version messages which are issued after an acceptable output has been achieved. Finally, the framework developed in the early sections of the chapter is extended to deal with versions that communicate via multiple ports.
5.1 Suitable Programming Paradigms

Choosing the principles of the programming paradigm on which a research or a new product will be based is one of the most important decisions to be made before design starts. The programming paradigm imposes implementation constraints and favours some design strategies whilst making others very difficult to realise. As a result, not only the implementation is shaped by the programming paradigm, but also the design. For example, in paradigms based on data encapsulation, e.g., object-oriented systems, one can assume that the data manipulated by the procedures of a module will not be altered by other modules directly. Thus, one must have a clear understanding of the desirable features of the programming paradigm before opting for a particular one.

Software fault-tolerance techniques must be complemented with mechanisms for tolerating hardware faults. Distributed architectures have been widely used in complex computing systems mainly to maximise available parallelism and to optimise the balance between communication and processing costs. In addition, they provide the necessary redundancy for implementing hardware fault-tolerance mechanisms. In short, "the fundamental properties of a distributed system are fault tolerance and the possibility to use parallelism" [Mul89]. Hence, research on software fault-tolerance must be conducted in distributed environments.

Fault-tolerance must be, as far as possible, orthogonal to the design, implementation, and testing of the functional components of the system. Functions which are to tolerate residual design faults, are more easily identified in systems that have been constructed based on abstractions [PT91] than in systems in which only one level of abstraction exists. Each abstraction may be a primitive or a composite component. Components must not have direct access to data managed by other components, because it would allow them to bypass fault-tolerance mechanisms and breach error confinement requirements. Thus, data must be encapsulated along with
the procedures that manipulate them directly in modules which can be designed, implemented, and tested separately. These modules are, then, assembled according to a design to form a system. A number of existing programming paradigms, such as *programming-in-the-small* versus *programming-in-the-large* [DK76, SKM85] and *object-oriented programming* [Cox86, Mey88], share these principles. These paradigms have the additional attraction of catering for the development of reusable software.

In order for software modules to be designed, implemented, and tested separately, they must not address other modules directly. This requirement is also necessary for modules to be utilised in various configurations without alteration. In programming environments that support indirect addressing, software modules define entry points, from which messages received from other modules are read, and exit points, where messages to be exported to other modules are written. Modules are assembled by linking exit points to entry points. Each software module may define several entry and exit points, because they may communicate with various software components at different points of their computation.

Throughout this dissertation, entry and exit points will be referred to by the name *port*. Four types of ports will be considered: *output*, *input*, *request*, and *reply* ports. Messages that do not require any response, henceforth referred to as *notification messages*, are written to output ports and read from input ports. Messages that demand a response, i.e., *request messages*, are written to request ports. Software modules read the *reply message* from the same request port to which the request message has been written. Server modules read request messages from reply ports and reply at the same reply port from which the request has been read.
5.2 Framework Realisation

Figure 5.1 depicts a fault-tolerant software module based on the framework outlined in figure 4.1, which is best realised in a programming environment that caters for data encapsulation, and indirect addressing. Fault-tolerant software component $\Sigma$ uses the complete configuration of the framework for designing, implementing, and testing software fault-tolerant components, henceforth referred to as SoFT, an acronym for Software Fault-Tolerance.

Throughout the dissertation, fault-tolerant modules, their ports and ports of their components are identified by Greek letters in contrast to single-version components and their ports, which are denoted by Latin letters. For the sake of clarity, Greek letters are not used to name components of SoFT modules, such as the versions and assessment components. Table 5.1 lists the meanings of the Greek letters.
that are usually utilised in representations of SoFT modules. Table 5.2 enumerates the acronyms used to represent components of the SoFT model.

External modules communicate with fault-tolerant module $\Sigma$ exactly as they would do if it were a primitive component. The types of ports $\xi$ and $\pi_1$ to $\pi_n$, via which controller $C$ communicates with external modules and the versions, respectively, have not been represented because they vary according to the role played by the SoFT module. Ports $\alpha_{\text{tu}}$, $\alpha_{\text{t}}, \alpha_{\text{ct}}$, and $\epsilon_0$ to $\epsilon_n$ are always request ports, whereas $\rho_1$ to $\rho_n$ are output ports in any configuration. The index zero refers to the universal error handling. Thus, $EHM_0$ and $\epsilon_0$ are the labels which identify the universal error handling module and the port to which the controller writes messages to it, respectively.

In Figure 5.1, $V_1$ and $CM_1$ are implemented as separate components. Thus, $V_1$ could be a module which has originally been designed to be used outside the SoFT framework. No link between $V_1$ and $CM_1$ is visible in figure 5.1, because this is not mandatory. However, they may communicate if it is convenient or necessary. Since $V_1$ and $CM_1$ are independent modules, they have separate threads of control. If $V_1$ needs to wait until the compensation procedure has been has completed by $CM_1$, they must synchronise explicitly. $V_2$ has been configured to read messages written to $\rho_2$. It suggests that $V_2$ has been specially designed to be used in a SoFT configuration, because messages written to $\rho_i, 1 \leq i \leq n$, are feedbacks on the status of the output produced by version $i$. Port $\rho_n$ is not linked, indicating, in this example, that $V_n$ does not require any compensation algorithm.

In server SoFT modules, error handling modules will probably instruct the controller to export a message to the external client reporting the failure of the assessment components to achieve an acceptable output. However, in other configurations, there is no obvious module to which to report the error. In fact, even in server SoFT modules, error messages, such as error logs, may need to be sent to modules other than the client. These messages can be exported from the SoFT module via ports
Chapter 5. A Software Fault-Tolerance Framework

<table>
<thead>
<tr>
<th>Letter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma$</td>
<td>name of the SoFT module</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>name of port by which the SoFT module communicates with other modules of the system</td>
</tr>
<tr>
<td>$\xi$</td>
<td>name of port by which the controller communicates with other modules of the system</td>
</tr>
<tr>
<td>$\alpha_{iv}$</td>
<td>name of port by which the controller communicates with the input validation module</td>
</tr>
<tr>
<td>$\alpha_s$</td>
<td>name of port by which the controller communicates with the voter</td>
</tr>
<tr>
<td>$\alpha_{ct}$</td>
<td>name of port by which the controller communicates with the conformity test</td>
</tr>
<tr>
<td>$\xi_1 \cdots \xi_n$</td>
<td>name of port by which the controller communicates with versions 1 to $n$</td>
</tr>
<tr>
<td>$\rho_1 \cdots \rho_n$</td>
<td>name of port by which the controller communicates with compensation modules 1 to $n$</td>
</tr>
<tr>
<td>$\epsilon_0 \cdots \epsilon_k$</td>
<td>name of port by which the controller communicates with error handling modules 0 (universal) to $k$</td>
</tr>
<tr>
<td>$\lambda_1 \cdots \lambda_m$</td>
<td>name of port by which an error handling module report the error to an external module</td>
</tr>
</tbody>
</table>

Table 5.1: Greek Letters used in SoFT Modules

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>Controller</td>
</tr>
<tr>
<td>$IVM$</td>
<td>Input Validation Module</td>
</tr>
<tr>
<td>$V$</td>
<td>Voter</td>
</tr>
<tr>
<td>$CT$</td>
<td>Conformity Test</td>
</tr>
<tr>
<td>$V_1 \cdots V_n$</td>
<td>Versions 1 to $n$</td>
</tr>
<tr>
<td>$CM_1 \cdots CM_n$</td>
<td>Compensation Modules 1 to $n$</td>
</tr>
<tr>
<td>$EHM_0 \cdots EHM_k$</td>
<td>Error Handling Modules 0 to $k$</td>
</tr>
</tbody>
</table>

Table 5.2: Acronyms used in SoFT Modules
\lambda_1 \cdots \lambda_m$, where $m$ is not necessarily equal to $k+1$, the number of error handling modules. More than one error handling module may be linked to the same $\lambda_i$, $1 \leq i \leq m$, port and a single error handling module may be linked to various $\lambda_i$ ports at the same time. Ports $\lambda_1 \cdots \lambda_m$ and the ports of the error handling modules linked to them are optional. For the sake of clarity, ports $\lambda_i$ are not usually represented in diagrams of SoFT modules throughout this thesis.

The components of the SoFT model may be either primitive or composite. SoFT is a recursive model, because its components may contain sub-components which are based on the SoFT architecture. Two examples of such configurations are depicted in figures 4.3 and 9.5.b. The former represents a fault-tolerant assessment component. The latter illustrates two versions that are implemented as SoFT modules.

### 5.2.1 Controller Functioning

The function of the controller is to receive messages from other SoFT components or from external modules and to forward them to the appropriate components. This control function is not unique for all possible configurations of the SoFT module. For example, the controller of a server SoFT module in which the outputs of the versions are set to be assessed in a first-come-first-served order has a different behaviour from that in which the outputs of the versions are assessed in a static order. Therefore, there must be either a different controller for each possible configuration of the SoFT model, or a single controller, or a reduced set of controllers which are capable of identifying the configuration of the SoFT module. This topic is further discussed in section 7.3. The controller is fully specified in section A.2.

As far as the underlying operating system is concerned, the components of the SoFT model are application components which have no means of controlling the execution of other components directly. The controller can, however, establish the execution mode of the versions indirectly by determining the times at which the messages received from external components are sent to the versions. Sequential
execution is achieved by sending the external message to the versions one at a time, following the order in which they have been enumerated. The external message is transmitted to a version only when the outputs of its predecessor have been assessed and if no reliable result has been agreed upon. Conversely, in order for the versions to execute concurrently, the controller sends the message that it has received from an external component to all the versions simultaneously. Naturally, only when the versions play the role of servers can the controller determine whether the versions will execute sequentially or concurrently. Thus producer, client, and consumer always execute in the concurrent mode, but the order in which their outputs will be assessed can be either static or dynamic.

In the diagram of figure 5.1, the versions have defined a single port to interact with the controller. Hence, the controller needs to create only one port, \( \xi \), to communicate with external components. However, versions that communicate via multiple ports are not rare and will be investigated in section 5.6. Notwithstanding the need to cater for multi-port versions, the operation of the SoFT framework can be explained for mono-port versions without loss of generality. It will be shown in section 5.6 that the mono-port controller, i.e., the controller that communicates with external modules through a single port, will be replaced by a number of concurrent mono-port controllers in configurations where the versions communicate via multiple ports.

## 5.3 Protocols

Transparent fault-tolerance implies that external modules should not need to be customised to communicate with SoFT components. They must be able to use the same protocol employed in interactions with non-fault-tolerant components. As the controller has no knowledge either of the syntax or of the semantics of the messages that it exchanges with external modules (section 3.5), it has to treat them purely as strings of bytes. As far as external modules are concerned, the controller emulates
Chapter 5. A Software Fault-Tolerance Framework

function iv_protocol( Msg_Type external_msg
  integer ehm_id, Msg_Type ehm_msg ): return Status_Type
request at αiv.a_msg( external_msg )
  case inexistent( ) →
    iv_protocol := SUCCESS;
  case success( ) →
    iv_protocol := SUCCESS;
  case failure( ehm_id, ehm_msg ) →
    iv_protocol := FAILURE;
  within iv.timeout;
end

Figure 5.2: Controller Interaction with the Input Validation Module

the behaviour of a functionally equivalent primitive component. For the same reason,
the controller regards the messages exchanged with the versions as strings of bytes,
whose syntax and semantics are irrelevant. As the design of the controller aims to be
as generic as possible, this protocol applies even when transparency is not desired,
i.e., when the versions are especially designed to be configured in a SoFT module,
e.g., exploiting compensation messages to promote internal error recovery.

5.3.1 Input Validation Protocol

The controller submits each message that it imports at port ξ to the input validation
module (IVM) before it forwards the message to the versions via ports π₁ \ldots πₙ. The
interaction between the controller and the input validation module is represented
in figure 5.2, which has been extracted from section A.2. Controllers of consumer
modules become ready again to receive new external messages. Controllers of modules
which are acting either as clients or as servers, initiate the output assessment
protocol, which is shown in section 5.3.2. An example of an input validation module
which verifies whether the set up value of a sensor lies within valid limits is shown
in figure 5.3, reproduced from section A.4.1.

The two possible input validation protocols are illustrated in figure 5.4. The
simple-component ivm()

reply-port service;

do true →
receive at service
  case a_msg(real value) →
    if within_range(value) = true →
      reply service.success();
    else →
      reply service.error();
    fi
  within eternity;
  od
end

Figure 5.3: Example of an Input Validation Module

output assessment state, in figure 5.4.b, is represented by a double circle in order to indicate a sequence of states, rather than a single one. This sequence is fully described in section 5.3.2. Should the status be incorrect, the reply message will contain, in addition to the assessment status, the index of the error handling module to be invoked by the controller, and any data that the error handling module might need to carry out its computation successfully.

5.3.2 Output Validation Protocol

Many interactions between the controller and the voter may be necessary before an agreement is reached. Assuming that the current state of all the versions is the same and that each of the versions issues a single message at each state, the voting protocol commences when the controller receives a message from one of the versions and terminates either when the voter reaches an agreement, or when it indicates that such an agreement is not possible, or when the deadline for providing a reliable output expires.

In the initiation phase of the voting protocol, the controller sends a message to
Chapter 5. A Software Fault-Tolerance Framework

Figure 5.4: Input Validation Protocol

```
function initiate_voter( ): return boolean
    request at α initiated msg( n_of_versions );
    case voter_exists( ) →
        initiate_voter := true ;
    case voter_does_not_exist( ) →
        initiate_voter := false ;
    within 0;
end
```

Figure 5.5: Initiation of a New Round of Voting

the voter informing it of the number of versions that have been configured. This message needs to be repeated at the initiation phase of each round of voting, because versions can be added or subtracted by re-configuration messages. Moreover, it is used to request the voter to start a new round of voting, because the controller may terminate a vote by time-out without notifying the voter. In addition, the voter initiation procedure enables the controller to learn whether a voter has been configured or not. No response may be given to the initiation message. The voter initiation procedure is depicted in figure 5.5, which has been copied from section A.2.

Following the initiation phase, the controller sends to the voter the message which has triggered the protocol and waits for a reply. The controller keeps reading messages from the versions and sending them to the voter, whilst the reply of the
voter indicates that it needs to assess more messages before reaching a decision. Whilst a vote is in progress, the controller elects to read only from the ports linked to the versions from which it has not yet received messages in the current round of voting.

The voting protocol is shown in figures 5.6, 5.7, and 5.8, all of which have been extracted from section A.2. Should the assessment order of the version outputs be static, the controller will import the version outputs in the same order in which the versions have been enumerated (figure 5.6). If the assessment order of the version outputs is dynamic, the controller will import the version outputs in a first-come-first-served order (figure 5.7). An example of a majority voter is listed in figure 5.9, which has been reproduced from section A.4.3.

When the voter outcome is successful, it returns an indication of the versions whose messages have agreed with the outcome of the voting, of those whose messages have disagreed, and of those whose messages have not yet been received by the voter. This list is used in the invocation of the compensation modules. Should the voter fail to reach a consensus, it returns an indication of the error handling module that the controller is to invoke, and any data that might be necessary for the error handling module to carry out its computation successfully.

The communication between the controller and the conformity test is similar to the communication between the controller and the input validation module. Figure 5.10, extracted from section A.2, depicts the algorithm used by the controller in this interaction. An example of a conformity test which verifies whether a value lies within valid limits is shown in figure 5.11, reproduced from section A.4.2.

The state diagrams of the protocols between the controller and the voter and between the controller the conformity test are shown in figures 5.12.a and 5.12.b, respectively. When the outcome of the voting is to be submitted to a conformity test, the protocol between the controller and the output assessment components
function static_voterjirst( Status_Type statusjof,
    Msg_Type successful_msg,
    integer ehm_id,
    Msg_Type error_msg ): return Status_Type

time time_out := eternity ;
integer index := 1;
Status_Type decision := INDECISION;
do index ≤ n_of_versions and decision = INDECISION → 
    receive at π[index]
    ...
    case communication_error( Communication_Status code ) →
        decision := voter_protocol_failure( index, ehm_id, ehm_msg );
        index := index + 1;
        if time_out = eternity ;
            time_out := version_timeout;
        fi
    case version_msg( Msg_Type version_msg ) →
        decision := voter_protocol_success( index, version_msg,
            status_of, successful_msg,
            ehm_id, ehm_msg );
        index := index + 1;
        if time_out = eternity ;
            time_out := version_timeout;
        fi
    within timeout;
    od
    ...
end

Figure 5.6: Voter Protocol with Static Output Assessment Order
function dynamic_voter_first( Status_Type status_of,
    Msg_Type successful_msg,
    integer ehm_id,
    Msg_Type error_msg ) : return Status_Type

    bag of port in := π[1] + \cdots + π[n.of.versions];
    time time_out := eternity ;
    Status_Type decision := INDECISION;
    do in ≠ ∅ and decision = INDECISION →
        receive at any of in
        ...
        case communication_error( Communication_Status code ) →
            decision := voter_protocol_failure( index_of( sender ), ehm_id, ehm_msg );
            in := in − sender ;
            if time_out = eternity ;
                time_out := version_timeout;
                fi
        case version_msg( Msg_Type version_msg ) →
            decision := voter_protocol_success( index_of( sender ), version_msg,
                status_of, successful_msg, ehm_id, ehm_msg );
            in := in − sender ;
            if time_out = eternity ;
                time_out := version_timeout;
                fi
        within timeout;
        od
        ...
    end

Figure 5.7: Voter Protocol with Dynamic Output Assessment Order
function voter_protocol_success( integer version_id,
    Msg.Type version_msg,
    Status.Type status_of,
    Msg.Type successful_msg,
    integer ehm_id,
    Msg.Type ehm_msg ): return Status.Type
request at a_v.version_msg( version_id, version_msg )
    case success( status_of, successful_msg ) →
        voter.protocol.success := SUCCESS;
    case indecision( ) →
        voter.protocol.success := INDECISION;
    case failure( ehm_id, error_msg ) →
        voter.protocol.success := FAILURE;
within voter.timeout;
end

function voter_protocol_failure( integer version_id,
    integer ehm_id,
    Msg.Type ehm_msg ): return Status.Type
request at a_v.fail_msg( version_id )
    case indecision( ) →
        voter.protocol.failure := INDECISION;
    case failure( ehm_id, error_msg ) →
        voter.protocol.failure := FAILURE;
within voter.timeout;
end

Figure 5.8: Controller Communication with the Voter
type Msg_Type = bag of universal;
type Status_Type = {SUCCESS or FAILURE or INDECISION};

simple-component voter()

reply-port service;

state
  integer n_of_versions;
  bag of Msg_Type version_msg;
  bag of Status_Type status_of;

do true ->
  receive at service
    case init_msg( integer n_versions ) ->
      n_of_versions := n_versions;
      version_msg := empty;
      for integer i := 1 to n_of_versions ->
        status_of[i] := INDECISION;
    af
    case fail_msg( integer version ) ->
      status_of[version] := FAILURE;
      if majority_possible( ) = true ->
        reply service.indecision( );
      else ->
        reply service.error( integer ehm_id := 0, Msg_Type ehm_msg :=
          "Communication error with most versions" );
    fi
    case version_msg( integer version, Msg_Type msg ) ->
      version_msg[version] := msg;
      status_of[version] := SUCCESS;
      if majority_found( ) = true ->
        comp_and_mark( the_majority( ) );
        reply service.success( status_of, Msg_Type the_majority( ) );
      elseif majority_possible( ) = true ->
        reply service.indecision( );
      else ->
        reply service.error( integer ehm_id := 0, Msg_Type ehm_msg :=
          "No agreement has been reached." );
      fi
      within eternity;
  od
end

Figure 5.9: Example of a Majority Voter
function ct.protocol( Msg.Type version_msg
    integer ehm_id, Msg.Type ehm_msg ): return Status_Type
request at a.ct.a_msg( version_msg )
case inexistent( ) — >
    ct.protocol := SUCCESS;
case success( ) — >
    ct.protocol := SUCCESS;
case failure( ehm_id, ehm_msg ) — >
    ct.protocol := FAILURE;
within ct.timeout;
end

Figure 5.10: Controller Interaction with the Conformity Test

simple-component ct()

reply-port service;

do true — >
    receive at service
    case a_msg( real sensor_reading ) — >
        if within_range( sensor_reading ) = true — >
            reply service.success( );
        else — >
            reply service.error( );
        fi
    within eternity ;
od
end

Figure 5.11: Example of a Conformity Test
is represented by the state diagram illustrated in figure 5.12.c. Finally, the state diagram shown in figure 5.12.d describes the protocol between the controller and the output assessment components, when the version outputs are to be submitted to a conformity test before being sent to the voter.

The diagrams depicted in figure 5.12 are valid for producer SoFT modules and other configurations where the input validation module is not present. When the input validation module has been configured, an extra state is introduced. In client SoFT configurations, when an acceptable output has been agreed upon by the voter and/or conformity test, the controller moves to an input validation state in order to assess the reply message before returning to the idle state. In server SoFT modules, the controller passes through this state, in order to assess the request, before initiating the assessment of the outputs.

5.3.3 Error Handling Protocol

The status of the assessment of the output produced by a version is sent to its associated compensation module when the controller receives the verdict of the output assessment components. The communication between the controller and the compensation modules (or the compensation procedures of non-transparent versions) is depicted in figure 5.13, copied from section A.2. An example of the general structure of a compensation module is shown in figure 5.14, extracted from section A.4.4. No message is sent to the compensation modules associated with versions whose outputs have not been assessed, e.g., late arrivals, or outputs from low priority versions which have not been needed. However, the index of these versions is stored along with the reliable output. This information will be necessary if these versions eventually issue their output. This topic is discussed in section 5.5.

An error handling module is activated by a request message that contains data which is need by the error handling module to carry out its computation successfully (sections 5.3.1 and 5.3.2). On its reply, the error handling module may indicate that
Figure 5.12: Output Assessment Protocols
function compensation_protocol( Status_Type status_of,
                          Msg_Type out_msg, Msg_Type in_msg )
    bag of integer late_versions := ∅;
    fail integer index := 1 to n_of_versions →
        if status_of[index] = SUCCESS →
            send ρ[index].success( );
        elseif status_of[index] = FAILURE →
            send ρ[index].failure( out_msg );
        elseif status_of[index] = INDECISION →
            late_versions := late_versions + index;
        fi
    af
        if late_versions ≠ ∅ →
            past_exports := past_exports + {out_msg, in_msg, late_versions};
        fi
end

Figure 5.13: Controller Interaction with the Compensation Modules

simple-component cm()

input-port feedback;

do true →
    receive at feedback
        case success( ) →
            (no compensation procedure);
        case failure( Msg_Type successful_output ) →
            (perform compensation procedure based on successful_output);
            within eternity ;
        od
end

Figure 5.14: Example of a Compensation Module
the data issued in the reply message is to be sent to:

- external modules
- the compensation modules (all of them)
- external modules and the compensation modules (all of them)
- no component
- another error handling module

The communication between the controller and an error handling module is represented in figure 5.15, which has been extracted from section A.2. In request-reply interactions initiated by an external element, for instance, the error handling module may indicate that its reply is to be sent to the external client that has initiated the interaction. This allows for the SoFT module to generate and send error indication messages that can be understood by the external client. An example of such an error handling module is illustrated in figure 5.16, which has been extracted from section A.4.5.

Another example of the application of messages issued by error handling modules arises in history-dependent versions that are linked to their respective compensation ports. The error handling module could generate a message that, sent to the compensation modules, would allow the versions to move to a safe state, since their computation has been invalidated by the assessment components without them agreeing on a correct output.

5.4 Errors in the Time Domain

When an error occurs in a faulty version, the version can send an output with an incorrect content, or fail completely to produce a message that is due at the time,
function ehm_protocol( integer ehm_id, Msg.Type error_msg )
request at e[ehm_id].error_handling( error_msg )

  case inexistent( ) →
    skip
  case external( Msg.Type ehm_reply_msg ) →
    reply e_.plain( ehm_reply_msg );
  case compensation( Msg.Type ehm_reply_msg ) →
    fa integer i := 1 to n.of.versions →
      send p[i].general_failure( ehm_reply_msg );
    af
  case both( Msg.Type ehm_reply_msg ) →
    reply e_.plain( ehm_reply_msg );
    fa integer i := 1 to n.of.versions →
      send p[i].general_failure( ehm_reply_msg );
    af
  case none( ) →
    skip
  case another_ehm( integer ehm_id.2, Msg.Type ehm_reply_msg ) →
    ehm_protocol( ehm_id.2, ehm_reply_msg );
within ehm_timeout;
end

Figure 5.15: Controller Interaction with an Error Handling Module

simple-component ehm()

reply-port service;

do true →
  receive at service
  case error_handling( Msg.Type error_msg ) →
    reply service.external( error_msg );
  within eternity ;
  od
end

Figure 5.16: Example of an Error Handling Module
or issue a message too late to be used. The first type of error, known as byzantine error, is detected by the output assessment components, namely: the voter and the conformity test. This section investigates algorithms used by the controller to detect the remaining two types, which are classified as errors in the time domain.

Errors in the time domain can only be detected if there is a deadline by which a particular event must have occurred. The controller utilises time-out clauses which impose deadlines for the reception of messages sent by the components that it communicates with. The value of the maximum time interval within which messages from each of the versions must be received by the controller is configured as a parameter when the controller is instantiated and can be changed by re-configuration messages.

The purpose of the use of time-out clauses is to prevent deadlocks, where the controller remains blocked indefinitely waiting for a component to send a message. In order to avoid incorrect error detections, the value of such clauses must take into account acceptable delays caused by heavy network traffic and node load. These time-out clauses, however, are not necessarily compatible with the real-time constraints of the external clients. In order to allow the SoFT module to be linked to different external modules with distinct real-time constraints at the same time, the responsibility for determining whether the SoFT module meets a particular real-time requirement rests on each of the external clients and not on the controller.

Designers must not overlook the response time of SoFT servers, which is, in general, expected to be greater than single-version servers. A SoFT module may comply with its functional specification, but fail to meet its real-time constraints.

In contrast to the external clients of server SoFT modules, which may perform different functions, the versions of a client SoFT module have the same real-time constraints. Thus, the real-time deadlines of the versions should be used as an upper bound for the time-out clauses used by the controller in interactions with external
servers. This is particularly useful when the response time of the external servers is not known.

5.4.1 Time-out Clauses in Communications with the Versions

An error in the time domain in one or more of the versions is detected when they fail to send a message to the controller within a pre-established deadline. In server configurations (section 6.1), the countdown for the deadline could start as soon as the external request has been forwarded to the versions. In this strategy, called absolute countdown, the designer must have a reasonable estimate of the time taken by the versions to process the request. However, since this value may vary significantly according to the input, the value of the time-out clause may be greatly overestimated or underestimated. In addition, it is necessary to take into account in the calculation of the time-out value acceptable delays caused by network traffic and node load.

It is reasonable to suppose that requests that take a long time to be processed by one of the versions will also take a long time to be processed by the others. Hence, the SoFT controller uses a different approach, whereby the countdown starts only when the first reply is received. Relative time counting, as this strategy is known, has the advantage of filtering common delays in the versions. Relative time countdown takes into account the delays of the versions in relation to one another, rather than absolute delays. Moreover, this is the only possible approach when the SoFT module acts as producer or client, because only when a message from one of the versions is received does the controller become aware that the remaining versions are supposed to issue a message.

In SoFT modules in which the versions execute concurrently and are assessed in a first-come-first-served order, if no reliable output has been agreed upon by the time the deadline expires, the controller can be sure that versions which have not yet sent their output will not do so in time. However, if the versions are assessed
in a pre-established order, it is possible that low priority versions have already sent their outputs by the time the deadline has expired, whilst the controller was waiting for the output of version $V_i$, $1 \leq i < n$. Thus, the controller regards version $V_i$ as faulty and proceeds reading the ports linked to $V_{i+1}$ to $V_n$ with a time-out clause equal to zero. As the deadline used by the controller to detect errors in the time domain is not a real-time one, the overhead in time of reading and assessing the outputs of $V_{i+1}$ to $V_n$ is acceptable. The value of the time-out clause ought to be calculated for the slowest of the versions.

In sequential configurations, the controller reads the message produced by version $V_j$, $2 \leq i \leq n$, where $n$ is the number of existing versions, only after the message sent by version $V_{j-1}$ has been assessed. Consequently, if the deadline expires whilst the controller is waiting for the message from version $V_k$, $1 \leq k < n$, it cannot be sure whether versions $V_{k+1}$ to $V_n$ will or will not produce an acceptable output. The controller must, then, send the original request to version $V_{k+1}$ and recommence the countdown, because version $V_{k+1}$ will not have received the external request. The controller may use the same time-out clause for all of the versions or distinct values for each of them, although the latter increases the number of configuration parameters of the controller.

### 5.5 Late Messages

In client SoFT modules, the controller ought to send either the external reply or an error indication to all the versions whose individual requests have been assessed. Those which have failed to send a request or those whose message does not match the reliable request have to be notified accordingly via their respective compensation module port. A copy of the external reply must be sent along with the compensation messages relative to the versions whose requests have not matched the reliable request in order to allow the versions to promote internal error recovery.
Enforcing the general rule enunciated above would be straightforward if all the versions had been assessed by the time the reply was received, or the waiting time expired. However, in most cases, the reliable output will be determined without analysing the messages from all the versions. In configurations based solely on the conformity test, a reliable request is achieved when one of the individual requests is validated. In SoFT modules using a voter, followed or not by a conformity test, the assessment of the outputs produced by the versions concludes when the voter has received enough messages to allow it to reach a consensus or to guarantee that such an agreement cannot be achieved.

Deciding upon a reliable request without assessing the messages from all the versions allows for the reliable request to be issued within the minimum possible time, but it leaves the controller with no knowledge of the versions whose requests have not been assessed. These requests could have been produced already and be either in the message queue of the controller waiting to be read, or in the network. It is also possible that versions which have not issued their messages by the time the controller receives the final decision of the assessment components still do so within the deadline. Naturally, some or even all of the remaining versions might miss the deadline, leading to errors in the time domain. Hence, the problem that arises from this situation is to identify, amongst the versions whose outputs have not been assessed by the voter and/or by the conformity test, those which have issued or will eventually issue their individual requests in time and those which will fail to do so.

This problem is not particular to the SoFT model. It occurs in any framework for implementing software fault-tolerant clients based on these assessment strategies. Sequence numbers cannot be adopted in these cases, because it is not a matter of detecting the loss of messages in the network. This approach would fail to detect that the version had missed a message, because the sequence number associated to a version increases only when the version issues a message. Although the solution proposed in section 5.5.1 is described in terms of its particular implementation in
the SoFT architecture, its principles are generic and it is believed that they can be applied to other models.

5.5.1 The Matching Voter

The controller must not send the reply received from an external server to a version unless it is sufficiently confident that such a version has produced a request which matches the one sent to the external server. A corollary of this rule is that the controller will always wait for the request to arrive before it sends the reply.

The controller is capable of identifying, amongst all the versions whose individual requests have been assessed before the reliable request has been reached, those which have produced messages that have agreed with the reliable output and those whose outputs have been rejected. Based on this knowledge, the controller will transmit the external reply to the versions whose messages have been validated and the appropriate compensation messages to those which have failed.

The controller cannot discard the external reply after it has been forwarded to all the versions whose outputs have been assessed, because there might be versions which must still receive it. Thus, before starting to receive a new request, the controller reads the ports which have not been inspected in order to verify whether there are messages in the queue. If this is the case, the controller submits each of them at a time to a matching procedure, the goal of which is to determine whether the message agrees with the previous request. If they match, the controller responds to the sender with the external reply. Otherwise, an error indication is sent to the compensation module associated to the sender version. Following this stage, the controller creates a list of pending requests which contains the reliable request, the external reply, and the indices of the versions whose requests have not yet been read by the controller. This list is necessary, because the missing messages may still arrive within the deadline established by the controller.
Whenever the controller reads a message, it searches the version index in the list of pending requests. If the index is not found, indicating that the message is a new one, the controller resumes the ordinary assessment procedure. Otherwise, the controller starts a matching procedure, because that could be a late message. Should the message agree with the former request, the external reply stored alongside the former request is sent to the version. An error message is sent otherwise to the associated compensation module. The index of the version is then removed from the list of pending requests. A past reliable request and its external reply are removed from the pending requests list in two circumstances: either when the outputs of all the versions have been examined or when the deadline used by the controller on that interaction expires. The latter implies that a timer must be used to eliminate nodes when deadlines lapse. When a node is eliminated from the list of pending requests, the controller sends a message indicating an error in the time domain to the compensation modules associated with the versions whose indices remained in the node. In addition, it sends a message containing these indices to an error handling module.

The matching procedure cannot always be done on a bit-by-bit basis. Since the matching criteria may be application dependent, the matching procedure cannot be realised by the controller. In some applications, the voter used in the normal assessment procedure could be applied, but if the new message arrived after a new interaction has initiated, it would require from the voter the ability to carry out two independent collations at the same time: a matching comparison with former requests and a voting amongst the current messages. Furthermore, dynamic voters, i.e., voters that use criteria that change at each new round of collation, cannot compare new and past messages, because the voting criterion could have changed when the past interaction had been concluded. Hence, client SoFT modules must have an extra element, referred to as the matching voter. Figure 5.17 depicts the general diagram of a client SoFT module. In a SoFT module where the voter is present, the matching voter may be a different instance of the assessment voter.
The matching voter is of greater importance for client SoFT modules, because late versions must receive the external reply, if they eventually issue their requests within a tolerable delay. However, the matching voter must be configured in producer and server SoFT modules in which a feedback on the status of the outputs of the versions is to be sent to the compensation procedures.

5.6 Multi-Port SoFT Modules

The analysis of the SoFT model made so far has assumed that the versions communicate via one single port. However, this configuration is seldom found in real-life applications. Therefore, the SoFT framework must be capable of dealing with versions that communicate through multiple ports.

One possible solution for this problem would be to modify the controller to use m
ports to communicate with external modules — where $m$ represents the number of ports used by each of the versions. This would mean that the controller would need $m \times n$ ports — where $n$ is the number of versions — to interact with the versions and the same number of ports, $m \times n$, to communicate with the compensation modules. However, this solution has two major drawbacks. Firstly, the multi-port controller, i.e., the controller that communicates with external components via multiple ports, must be able to deal asynchronously with the messages from external and internal components. Secondly, internal messages issued at different ports might need to be assessed by different conformity tests and voters. Thus, the controller would need $m$ ports to communicate with up to $m$ different conformity tests, and another set of $m$ ports to interact with distinct voters. Moreover, the design of such a controller would be much more complex than the one introduced in chapter 5, possibly leading to controllers that could not be regarded as much more reliable than the versions.

A corollary of the previous analysis is that the algorithm of a controller which deals with multi-port versions is essentially a concurrent one. Thus, as the SoFT model assumes that the underlying operating system supports multi-tasking, and as the messages received at each of the ports linked to the same version do not need to be cross examined, the multi-port controller can be implemented as a composite component made of mono-port controllers which execute concurrently, henceforth referred to as *concurrent controller*. Mono-port controllers use a single port to communicate with external components. Each of the mono-port controllers is responsible for exchanging messages with one of the ports of each of the versions. Needless to say, the version ports must be equivalent. This approach allows for the use of different input validation modules, voters, conformity tests, and error handling modules for each of the ports, if necessary, without changing the design of the controller depicted in figure 5.17. Each of the mono-port controllers is configured independently. Thus, for example, those which deal with input messages only do not need to have the conformity test or the voter linked to it. Conversely, mono-port controllers which receive output messages that do not require a reply, do not need
to have the input validation module defined.

Filters are examples of modules that use more than one port to interact with other system components. A filter receives the outputs of its predecessor in the pipeline configuration at one port and exports the result of the filtering to its successor through another port. A filter SoFT module, labelled \( \Phi \), is illustrated in figure 5.18. The diversely designed versions of the filter, \( V_1 \cdots V_n \), occupy the central part of the picture.

To the left of the versions, there is the controller which receives the filter inputs. In this example, \( C_{in} \) uses the input validation module \( IVM \) to validate the messages that it receives at local port \( \xi \). Should no input validation be necessary, \( C_{in} \) could have been omitted and port \( \phi_{in} \) would have been linked to ports in of versions \( V_1 \cdots V_n \) directly. If the filter were to reply to its predecessor, a voter and/or a conformity test could have been configured. In the absence of both assessment components, \( C_{in} \) would forward, via local port \( \xi \), the first reply received from the versions. A controller \( C_{in} \) operating in the sequential mode would reply to the preceding module with the message received from the lowest index version that arrived within the established deadline.

The right hand side of figure 5.18 depicts controller \( C_{out} \), which uses voter \( V \) and conformity test \( CT \) to assess the outputs of the filter versions and error handling modules \( EHM_0 \cdots EHM_k \). No matching voter is needed in this case, because versions \( V_1 \cdots V_n \) do not require the external module that succeeds \( \Phi \) in the pipeline configuration to reply. For the sake of clarity, the compensation modules have been omitted from the picture.

### 5.6.1 Synchronisation of the Mono-port Controllers

The mono-port controllers of a SoFT module are independent from each other and need to synchronise only when a message is to be sent to the versions. This is
necessary to prevent the versions from changing to non-equivalent states. This mis-
behaviour may occur when two or more mono-port controllers send their messages to
the versions simultaneously. Due to process scheduling, for example, one controller
may be interrupted whilst transmitting to the versions and resume its computation
only after another controller had sent its message to all the versions. As a result, the
versions to which the former controller had sent its message prior to the interruption
could have moved to a particular state, whereas the versions that have received the
message from the latter controller first could have gone to a different state.

The necessary degree of availability of the SoFT controller (and the assess-
ment components and the error handling modules) is achieved by fault-tolerance meth-
ods based on identical replicas, as discussed in section 6.4. This approach, which
has been proposed for mono-port controllers, equally applies to concurrent ones.
A concurrent controller is made tolerant to hardware failures by replicating all of
the mono-port controllers that compose it. However, each replicated concurrent
function get_token( )
    request at τ_request.get_token( )
    case token_administrator.inexistent( ) →
        skip ;
    case token_granted( ) →
        skip ;
    within eternity ;
end

function free_token( )
    send at τ_free.free_token( );
end

Figure 5.19: Controller Algorithms for Token Allocation and Release

controller must execute at the same node.

The mono-port controllers use a token-pass protocol to synchronise. The token is managed by a token administrator, which is an independent component located at the same node. Since all the mono-port controllers and the token administrator execute in the same machine, there is no need to cater for token loss detection and token recovery due to hardware failures. As the controllers, by definition, have an acceptable degree of reliability, it is postulated that they will execute the token-pass protocol correctly. The same assumption can be made about the token administrator as a result of its simple design.

Figure 5.19 depicts the allocation and release of the token by the controller. Figure 5.20 lists the algorithm of the token administrator. Both specifications have been extracted from section A.2. No data need to be exchanged between the token administrator and the mono-controllers in order to implement the token passing protocol. The message exchange in this protocol acts as a binary semaphore [MD74, HGLS78, Dei84]. The token is granted on a first-in-first-out basis. The FIFO list is the queue of incoming messages at port token_request.

A simplified diagram of the configuration of the mono-port controllers and the token administrator is depicted in figure 5.21. The mono-port controllers commu-
simple-component token_administrator( )

    reply-port token_request;
    input-port token_release;

do true →
    receive at token_request
    case get_token( ) →
        skip;
    within eternity;
    reply token_request.token_granted( );
    receive at token_release
    case free_token( ) →
        skip;
    within eternity;
    od
end

Figure 5.20: Algorithm of the Token Administrator

The controller blocks until it receives a reply from the token administrator. When it finishes the transmission to the versions, it sends a message releasing the token to the token administrator through the second port, which is an output port.

The filter configuration shown in figure 5.18 does not require controllers $C_{in}$ and $C_{out}$ to be synchronised, as only $C_{out}$ sends messages to the versions. Synchronisation is only necessary amongst controllers that transmit messages to the versions.

5.7 Summary

A framework for developing fault-tolerant software components based on design diversity has been presented in this chapter. Fault-tolerant components are composite components that encapsulate dissimilar solutions of the problem (versions) and specially designed elements which interact with the versions in order to ensure a correct
external behaviour, in spite of possible faults in one or more of the versions. The outputs of the versions are assessed by a voter and/or a conformity test. Messages sent as inputs to the versions are inspected by an input validation module. Should a version fail to produce an acceptable output, a compensation module associated with it is invoked with the result of the assessment of the outputs of the remaining versions. It enables the amendment of external actions that have been taken by the version prior to the assessment of its output. If the assessment components are unable to agree on an correct output, an error handling module may be called in order to provide local mechanisms for dealing with the error. The versions, assessment components, compensation, and error handling modules interact under the co-ordination of a generic component called the controller. SoFT is a recursive model. Its components may be composite components that contain software fault-tolerant modules based on this framework.

The use of time-out clauses by the controller to detect errors in the time domain has been explained in this chapter. Another issue that has been investigated is the
problem of replying to client versions which have their outputs assessed after the reliable output has been achieved by the assessment components. As a result, a new component, namely the matching voter, was added to the original configuration of the SoFT model.

It was shown that to deal with versions that use more than one port to interact with other modules, the SoFT framework utilises a multi-port controller made of concurrent mono-port controllers. Each of these mono-port controllers communicate with one of the ports of the versions. In order to prevent the versions from moving to different states due to the concurrent execution of the mono-port controllers, a simple token passing protocol is used to synchronise the mono-port controllers that send messages to the versions.

A prototype of the framework proposed in this chapter and the modules of the experiment described in chapter 8 have been implemented in a programming environment based on the *programming-in-the-small* versus *programming-in-the-large* (appendix C). There is no practical experience with object oriented systems, but due to the similarities between these two paradigms, difficulties in realising the SoFT model in an object-oriented environment are not anticipated.
Chapter 6

Analysis of the SoFT Model

The SoFT model (reproduced in figure 6.1) is analysed in this chapter. The basic configurations of the SoFT model in terms of the roles that it might play are explained and exemplified. In addition, it is shown how different types of components and software fault-tolerance techniques can be realised in the framework. This is followed by the analysis of the complexity of messages exchanged in interactions of SoFT modules with other modules in the system. The need to provide hardware fault-tolerance to the SoFT model is also addressed in this chapter. Finally, the utilisation of the SoFT model as a testing platform is discussed.

6.1 Basic Configurations

Designers are usually concerned with unreliable services. Thus, software fault-tolerance has been mostly studied for server modules. However, faulty clients and producers can also be responsible for system misbehaviour. Although errors in consumer modules provoked by incorrect input messages may not be detectable at the time that they occur, they may have severe consequences in the future when the module issues a message or acts upon an external device. Therefore, not only servers, but also clients, producers, and consumers must be made fault-tolerant, if their degree of reliability is not satisfactory.
Filters can be regarded as consumers which become producers after they have processed the inputs received when they acted as consumers. Therefore, it suffices to show how the SoFT framework can be used to attach fault-tolerance to the remaining four types of modules enumerated in section 3.1, namely:

- producer
- consumer
- client
- server

There is no need for the input validation module in SoFT components which act as producers. At least one assessment component must be used to validate the outputs. Compensation modules and error handling modules are configured according
to the fault-tolerance requirements of each individual application. A configuration of a producer SoFT module is represented in figure 6.2. Messages sent by the versions, are received by the controller at ports $\pi_i \ (1 \leq i \leq n)$. They can be assessed either by voting only, or by the conformity test alone, or by a combination of a voter and a conformity test (section 4.2.1). The controller determines whether the voter is present or not by inspecting the return code of the primitive used to send the initiation message (section 5.3) to the voter. If there is no voter, the controller assumes that the messages are to be assessed only by the conformity test. Should the voter be configured at a later stage, the controller will be aware of its existence when it sends another initiation message. The high level specification of a producer SoFT controller is depicted in figure 6.3, extracted from section A.2.

Figure 6.4 depicts a consumer software fault-tolerant module based on the SoFT framework. The high level specification of a consumer SoFT controller is illustrated in figure 6.5, which has been reproduced from section A.2. Since consumer modules do not export messages, voters and conformity tests are not used in this type of module. Compensation modules are rarely defined in consumer SoFT modules, because there is no assessment of the internal computation of the versions. They are configured only if an error indication by the input validation module may result in calling an error handling module which, in turn, will issue a message to be delivered to the compensation modules. When an external message is received by the controller, it is sent to the input validation module for inspection. If no input validation module has been configured, the controller receives a return code from the communication primitive indicating that there is no component linked to the respective port. As a result, it regards the message as valid and sends it to all versions.

Client components export request messages and import replies. Therefore, at least the voter or the conformity test must be configured in client SoFT modules. The input validation module is optional. As in any configuration of the SoFT model, the instantiation of compensation modules and error handling modules depend on the recovery specification of the application. Figure 6.6 illustrates a client SoFT
function producer( )
    boolean voter_present := initiate_voter( );
    Status.Type out_assess := conc_out.protocol( voter_present,
        Status.Type status_of,
        Msg.Type successful_msg,
        integer ehm_id,
        Msg.Type error_msg );

    if out_assess = SUCCESS →
        send $\xi$.plain ( Msg.Type successful_msg );
        compensation.protocol( status_of, successful_msg, Msg.Type reply_msg := 0 );
    elseif out_assess = FAILURE →
        ehm.protocol( ehm_id, error_msg );
    fi
end

Figure 6.2: General Structure of a Producer SoFT Module

Figure 6.3: High Level Specification of a Producer Controller
function consumer( )
    receive at ζ
        case communication_error( Communication_Status code ) →
            ehm_protocol( integer ehm_id := 0,
                Msg_Type error_msg := “Communication error”+code+“ at ζ” );
        case communication ok( Msg_Type external_msg ) →
            Status_Type iv.status := iv.protocol( external_msg, ehm_id, error_msg );
            if iv.status = FAILURE →
                ehm_protocol( ehm_id, error_msg );
            elseif iv.status = SUCCESS →
                get_token( );
                fa integer i := 1 to n_of_versions →
                    send π[i].plain ( external_msg );
                af
                    free_token( );
            fi
        within eternity ;
    end

Figure 6.4: General Structure of a Consumer SoFT Module

Figure 6.5: High Level Specification of a Consumer Controller
module in which requests, produced by the versions, are assessed both by a voter and by a conformity test. Besides, this SoFT module utilises an input validation module to validate replies received from external servers. After exporting the request, the controller blocks, waiting for a reply. As there may be more than one server, the controller will unblock when the first reply arrives. The high level specification of a client SoFT controller is shown in figure 6.7, copied from section A.2.

A server based on the complete configuration of the SoFT framework is exemplified in figure 6.8. In this example, the individual replies of the versions are assessed both by a voter and by a conformity test. However, the minimal configuration of a server SoFT module comprises the versions, the controller and one of the components used to assess output messages. In fact, when only timing errors are expected, even the assessment components can be absent. The high level specification of a server SoFT controller is represented in figure 6.9, extracted from section A.2.
function client( )
  boolean voter_present := initiate_voter( );
  Status_Type out_assess := conc_out_protocol( voter_present,
    Status_Type status_of,
    Msg_Type successful_msg,
    integer ehm_id,
    Msg_Type error_msg );
  
  if out_assess = SUCCESS →
    request ξ .plain ( Msg_Type successful_msg )
    case communication_error( Communication_Status code ) →
      ehm_protocol( integer ehm_id := 0,
        Msg_Type error_msg := “Communication error” + code + “ at ξ” );
    case communication_ok( Msg_Type reply_msg ) →
      Status_Type iv_status := iv_protocol( reply_msg, ehm_id, error_msg );
      if iv_status = FAILURE →
        ehm_protocol( ehm_id, error_msg );
      elseif iv_status = SUCCESS →
        get_token( );
        for integer i := 1 to n_of_versions →
          reply ξ[i].plain ( reply_msg );
        af
        free_token( );
      fi
    within ext_timeout;
    compensation_protocol( status_of, successful_msg, reply_msg );
  elseif out_assess = FAILURE →
    ehm_protocol( ehm_id, error_msg );
  fi
end

Figure 6.7: High Level Specification of a Client Controller
Figure 6.8: General Structure of a Server SoFT Module
function server( )
receive at $\xi$

case communication_error( Communication_Status code ) →
    ehm_protocol( integer ehm_id := 0,
               Msg_Type error_msg := "Communication error"+code+" at $\xi$ " );
case communication_ok( Msg_Type external_msg ) →
    Status_Type iv_status := iv_protocol( external_msg, ehm_id, error_msg );
if iv_status = FAILURE →
    ehm_protocol( ehm_id, error_msg );
elseif iv_status = SUCCESS →
    if execution_mode = SEQUENTIAL →
        Status_Type request_assess := seq_req_protocol( Msg_Type external_msg,
                   Status_Type status_of,
                   Msg_Type successful_reply,
                   integer ehm_id,
                   Msg_Type error_msg );
    elseif execution_mode = CONCURRENT →
        Status_Type request_assess := conc_req_protocol( Msg_Type external_msg,
                   Status_Type status_of,
                   Msg_Type successful_reply,
                   integer ehm_id,
                   Msg_Type error_msg );
    fi
    if out_assess = SUCCESS →
        reply $\xi$_plain ( Msg_Type successful_reply );
        compensation_protocol( status_of, successful_reply, Msg_Type ext_msg := 0 );
    elseif out_assess = FAILURE →
        ehm_protocol( ehm_id, error_msg );
    fi
fi
within eternity ;
end

Figure 6.9: High Level Specification of a Server Controller
In single-version client-server configurations, the server is usually supposed to reply with an error message, if the service requested by the client cannot be completed. If replies containing error reports are acceptable to the assessment components, they are exported to the external client. For example, when most of the versions reply with an identical error message, a bit-by-bit majority voter does not distinguish it from a successful reply. As far as the voter is concerned, most of the versions have replied with the same message and, therefore, this should be the message to be exported to the external client. The fact that it is actually an error message is irrelevant both to the voter and to the controller. Error messages generated by error handling modules (section 4.3.3) result from the impossibility of the assessments components to agree upon a reliable output.

### 6.2 Traditional Software Fault-Tolerance Techniques in SoFT

The SoFT model caters for the combination and/or augmentation of important software fault-tolerance techniques. However, there are situations in which the best approach is to use these techniques in their canonical configuration.

Section 2.4 presents *conversation* as a technique for structuring communicating concurrent programs in order to avoid the domino effect caused by contamination amongst them. In the SoFT approach, the risk of contamination is reduced to the minimum possible. Since all messages exchanged with the diversely designed versions are assessed to the best of the ability of the assessment components, erroneous messages which have filtered through the assessment components could only be detected if the assessment components were redesigned, or prevented if the faults which caused them were fixed. Thus, the structure of the SoFT framework provides the same error insulation as conversations do.

This section discusses the realisation under the SoFT framework of the remaining
Chapter 6. Analysis of the SoFT Model

four well-known software fault-tolerance techniques presented in section 2.4, namely:

- compensation
- exception handling
- recovery blocks
- N-version programming

6.2.1 Compensation

The controller provides each of the versions with a feedback on their outputs along with the reliable output, if it has been achieved (section 5.3.3). A SoFT module based purely on compensation, however, contains a single version. Therefore, its associated compensation module will receive the status code only. The assessment of its outputs has to be carried out by a conformity test, as the SoFT module has only one version. Thus, a SoFT module based on compensation consists of the controller, the conformity test, one version, and its associated compensation module. When the SoFT module acts as a server, the input validation module may be configured. This configuration is depicted in figure 6.10. Compensation and normal computation procedures are implemented in separate components in this figure, but both could be encapsulated in the same component. The component could use two different ports to communicate with $C$ in order to distinguish between normal requests and compensation messages.

One could argue that an error handling module could play the role of the compensation module and, therefore, suggest that compensation modules are redundant elements in the SoFT model. Although this may be true for a SoFT module purely based on compensation, compensation modules and error handling modules have distinct roles in the complete model. The former are used to deal with errors in
6.2.2 Exception Handling

A server SoFT module based on exception handling is illustrated in figure 6.11. The assessment of the version outputs is done by a conformity test, because there is only one version in the SoFT module. The conformity test is not mandatory. If it does not exist, the SoFT module will rely solely on the internal mechanisms of the version for error detection. Interface exceptions are raised by the input validation module. Normal computation exceptions can be raised either by the version itself, or by the conformity test. Exceptions raised by the version can result from an internal error detection or from an error indication from lower level components, since the version can be a composite component. An exception is treated by one of the error handling modules, $EHM_0, \ldots, EHM_k$. If it does not succeed, it must ensure that the
SoFT module fails graciously and must indicate the error to the external client. In addition, it may perform other local tasks, such as the updating of an error logging file.

Exception handlers can only be implemented as independent components provided that they do not need to share state with the normal computation code. Exception handlers that need to correct internal variables of the normal code must be encapsulated in the same module as the normal code. The version must create as many ports as are internal exception handlers so that the exception handlers can be invoked by messages written by the controller to ports $\epsilon_0$ to $\epsilon_k$. 

Figure 6.11: Example of a SoFT Server based on Exception
6.2.3 Recovery Blocks

In order to implement a canonical recovery block [Ran75], the controller is configured to send external requests to the versions sequentially and to transmit their replies for assessment in a static order. A version needs to execute only if the previous one in line fails the acceptance test.

The principles on which the canonical recovery block structure is based are supported in the SoFT model. The conformity test, for instance, has been incorporated in the SoFT model to perform the same function as the acceptance test in the recovery block approach. The recovery block alternatives are implemented in SoFT as versions. Recovery blocks indicate an error to the calling component when all alternatives fail. In SoFT, error messages are generated by error handling modules. Hence, at least one of these components must be configured if an error report is required. Figure 6.12 depicts a SoFT server based on the recovery block approach. This recovery block has \( n \) "alternatives" implemented as versions \( V_1, \ldots, V_n \).

6.2.4 N-Version Programming

N-version programming [Avi85] is very easily mapped on to the SoFT model. Apart from the versions, the only other component that must be linked to the controller is the voter. In addition, an error handling module can be configured if an error indication is required. An example of a typical SoFT server based on N-version programming is illustrated in figure 6.13. No error handling module has been configured in this example. The controller sends external messages to the versions concurrently and transmits their outputs to the voter in a first-come-first-served order.

The architecture of client and producer SoFT modules based on N-version programming is a mirror-image of the one shown in figure 6.13. Client versions communicate with the controller via request ports and producer versions via output ports. In addition, controller ports \( \xi \) and \( \pi_1 \) to \( \pi_n \) are inverted. The versions execute
Figure 6.12: Example of a SoFT Server based on the Recovery Block Scheme

Figure 6.13: Example of a SoFT Server based on N-version Programming
concurrently, since the controller cannot determine the time at which the versions will produce their outputs.

6.3 Complexity of Messages in SoFT Modules

A notification interaction between a SoFT module and another module of the system is triggered by a message sent by one of them to the other and terminated when the message is received by the addressee. A request-reply interaction between a SoFT module and another module of the system is initiated by a request message sent by one of the modules (which acts as a client) to the other (which plays the role of the server) and terminated when a reply issued by the server is received by the client. Within an interaction, a number of messages are exchanged between the controller and the other components of the SoFT framework.

6.3.1 Complexity of Messages at Each Controller Port

The number of messages exchanged between external components and the controller ($N_\xi$) is either one, if the interaction is of the notification type, or two, if it is a request-reply interaction.

The number of messages exchanged between the controller and the versions ($N_{\sum_{i=1}^{n} \xi_i}$) varies according to the output assessment strategy that has been adopted and according to how successful the version outputs are. The minimum number of such messages is two. This situation arises when the versions play the role of servers, the output assessment is purely based on a conformity test, the execution mode of the versions is set to sequential, and the highest priority version passes the conformity test. In the worst case, in which the versions act as clients or servers, and the execution mode of the versions is set to concurrent, the number of messages exchanged between the controller and the versions is $2 \times n$, where $n$ is the number
of versions that have been configured.

When the assessment components reach a final decision, be it and agreement or not, the controller sends a single compensation message through ports $\rho_i$, $i = 1, \ldots, n$, where $n$ is the number of versions that have been configured. Thus, assuming that for each version $i$, either the version itself or its associated compensation port is linked to controller port $\rho_i$, or no link with $\rho_i$ exists, the number of compensation messages issued by the controller $\sum_{i=1}^{n} \rho_i$ may range from none up to $n$.

The input validation module receives a request from the controller and, as a result, it issues a reply validating or not the input. Thus, the number of messages exchanged between the controller and the input validation module $\sum_{i=1}^{n} \rho_i$ is always equal to two.

The number of messages exchanged between the controller and the conformity test $\sum_{i=1}^{n} \rho_i$ varies from two, if the first version output to be accessed is validated, to $2 \times n$ (where $n$ is the number of versions that have been configured), if all or all but one versions fail the conformity test.

The number of messages exchanged between the controller and the voter $\sum_{i=1}^{n} \rho_i$ varies according to the voting algorithm. Assuming that the minimum number of such messages is obtained when a majority voter is used, and the highest priority versions agree with one another, the minimum number of messages exchanged between the controller and the voter is $1 + 2 \times \left[ \frac{n+1}{2} \right]$, where $n$ is the number of versions that have been configured. The constant 1 in the expression refers to the initiation message. In the worst case, in which the outputs of all versions are needed for an agreement to be reached, or when no agreement is reached, the number of messages between the controller and the voter is $1 + 2 \times n$.

Each communication between the controller and the matching voter needs five messages: an initiation message, a message containing the past reliable output, an indecision indication sent by the matching voter as a result of such a message, a
message containing the late output, and the final reply from the matching voter. For any interaction between an external module and a SoFT module, the number of times that the matching procedure is required is, at most, equal to the number of versions whose outputs have not been assessed before the controller has terminated the interaction. If all the versions have had their outputs assessed before the interaction has terminated, the number of messages exchanged between the controller and the matching voter \( N_{\text{m,v}} \) is equal to zero. In the worst case, assuming that a majority voter is used, and the highest priority versions agree with one another, the number of messages exchanged between the controller and the matching voter is \( 5 \times (n - \left\lceil \frac{n+1}{2} \right\rceil) \), where \( n \) is the number of versions that have been configured. It's important to note however that, whenever the execution mode of the versions is set to concurrent, the best case in terms of the number of messages exchanged by the controller with the conformity test and the voter implies in the worst case in terms of the number of messages exchanged between the controller and the matching voter, and vice-versa.

Finally, the number of error handling modules that might be invoked in an interaction between an external module and a SoFT module may vary from zero, when no unrecoverable error occurs, to \( k + 1 \), which is the number of error handling modules that have been configured. Thus, since each interaction between the controller and an error handling module requires two messages (a request and a reply), the number of messages exchanged between the controller and the error handling modules \( \sum_{i=0}^{k} 2 \) varies from zero, in the best case, to \( 2 \times (k + 1) \), in the worst case.

### 6.3.2 Global Complexity of Messages

In a SoFT module in which all components have been configured, the number of messages exchanged within an interaction between an external module and the SoFT module \( N_{\text{SoFT}} \) is expressed by:
Chapter 6. Analysis of the SoFT Model

\[ N_{\text{SoFT}} = N_\xi + \sum_{i=1}^{n} N_{\pi_i} + \sum_{i=1}^{k} N_{\alpha_{ci}} + N_{\alpha_i} + N_{\alpha_{ci}} + N_{\alpha_{ci}} + \sum_{i=0}^{k} c_i \] (6.1)

The different configurations of the SoFT model that may influence the complexity of the messages exchanged between external modules and SoFT modules are depicted in table 6.1. This table indicates the number and complexity of messages exchanged between an external module and a SoFT module in two extreme situations: when no error occurs, and when the assessment components cannot agree on an acceptable value. In the latter, the outputs of all versions need to be assessed by the output assessment components before they indicate the failure. The voting algorithm in this analysis is based on majority voting. The evaluation of each configuration is described in appendix B.

The successful case is the usual one, because errors are expected to be rare. Thus, one can assume that the complexity of messages in the successful case as being the normal case.

In the analysis of the failure case, it is assumed that all \( k + 1 \) error handling modules will be invoked and, as a result of it, an error message will be sent to all \( n \) compensation modules (or to the versions, if they are linked to ports \( \rho_i \)). In addition, in configurations where the version outputs are assessed by the conformity test and then voted, it is assumed that the version outputs pass the conformity test. If they are rejected by the conformity test, the number of messages in the failure case is \( 2n + 1 \) less than the indicated expression, because the voter is never invoked. Furthermore, in configurations where the version outputs are assessed by the voter first and then, if the voter reaches an agreement, such an agreement is submitted to the conformity test, it is assumed that the voter needs to assess the outputs of all the versions in order to reach an agreement. The outcome of the voting, however, does not pass the conformity test. In server configurations, the invocation of the error handling modules results in a reply to the external client indicating that an
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Success</td>
</tr>
<tr>
<td></td>
<td>$n + 1 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$n + 3 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$6n - 3 \left[ \frac{n+1}{2} \right] + 2 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$6n - 3 \left[ \frac{n+1}{2} \right] + 4 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$7n - 1 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$7n - 3 \left[ \frac{n+1}{2} \right] + 3 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$7n - 3 \left[ \frac{n+1}{2} \right] + 5 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$7n + 1 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$7n - 3 \left[ \frac{n+1}{2} \right] + 3 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$7n - 3 \left[ \frac{n+1}{2} \right] + 5 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$7n + 1 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$7n - 3 \left[ \frac{n+1}{2} \right] + 5 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$6 \Rightarrow \mathcal{O}(1)$</td>
</tr>
<tr>
<td></td>
<td>$4 \left[ \frac{n+1}{2} \right] + 3 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$6 \left[ \frac{n+1}{2} \right] + 3 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$4 \left[ \frac{n+1}{2} \right] + 5 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$8 \Rightarrow \mathcal{O}(1)$</td>
</tr>
<tr>
<td></td>
<td>$4 \left[ \frac{n+1}{2} \right] + 5 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$6 \left[ \frac{n+1}{2} \right] + 5 \Rightarrow \mathcal{O}(n)$</td>
</tr>
<tr>
<td></td>
<td>$4 \left[ \frac{n+1}{2} \right] + 7 \Rightarrow \mathcal{O}(n)$</td>
</tr>
</tbody>
</table>

Table 6.1: SoFT Configurations for Message Complexity Analysis
unrecoverable error has occurred. These assumptions lead to the worst possible scenario in terms of message exchange in the failure case.

From the complexities depicted in table 6.1, one concludes that, regardless of the configuration of the SoFT model, the complexity of the messages exchanged between external modules and SoFT modules are never greater than $O(n)$, in the success case, and $O(n + k)$, in the failure case. This implies that the number of messages exchanged between external modules and SoFT modules grows linearly with the number of versions.

The expressions that appear in table 6.1 are evaluated for configurations in which the number of versions ($n$) is equal to three and in which only the universal error handling modules has been configured (i.e., $k + 1 = 1$). These figures are depicted in table 6.2. The small difference between most success and failure figures (in some cases, the number of messages in success cases are even higher than their failure counterparts) is caused by the relative great number of messages that are required in the matching voting procedure (which is equal to five). Since the outputs of all versions will have been assessed when a failure is indicated, no matching procedure occurs in the failure case. The relative high cost of matching procedures in terms of messages is the price that is paid to allow for acceptable outputs to be exported from the SoFT module as soon as they have been agreed on by the output assessment components.

The only class in which the number of messages in the success case is significantly higher than in the failure case is that of servers in which the versions execute sequentially. In this class, the matching procedure is not used in the success case because, after the output assessment components have agreed on an acceptable output, the external request is not forwarded to the remaining versions.
### Chapter 6. Analysis of the SoFT Model

#### Table 6.2: Message Complexity Analysis for $n = 3$ and $k = 0$

<table>
<thead>
<tr>
<th>Configuration</th>
<th>$N_{\text{SoFT}}(n = 3, k = 0)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>role</strong></td>
<td><strong>exec. mode</strong></td>
</tr>
<tr>
<td>cons.</td>
<td>conc.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>prod.</td>
<td>conc.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>client</td>
<td>conc.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>server</td>
<td>conc.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>seq.</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 6. Analysis of the SoFT Model

6.4 Availability Issues

The physical location of the components of the SoFT framework is not specified at the time that they are designed. Their actual mapping on to the nodes of the network is done when the application system is configured. The distribution of the components must ensure that hardware failures do not jeopardise the increase in software reliability achieved by the SoFT model.

In distributed systems, availability is maximised by configuring redundant modules in separate nodes and sequential ones in the same machine. Two or more modules are said to be redundant if they perform equivalent functions without interacting with each other, whereas two modules are regarded as sequential if all or part of the outputs of one of them are directly or indirectly used as inputs to the other.

Clearly, the versions are redundant components. Although their physical location does not affect the outcome of the assessment components, creating them in the same machine makes the SoFT module vulnerable to hardware errors. Hence, the versions must be located at different nodes of the network. For the same reason, each of the compensation modules ought to be instantiated in a separate machine. However, since a compensation module has a sequential relationship with its respective version, both of them must execute on the same computer. The controller has a client-server relationship with the assessment components and the error handling modules. Thus, the controller, the assessment components, and the error handling modules ought to be configured in the same node if availability is to be maximised.

Configuring each version-compensation module pair at a different machine provides tolerance to hardware faults in these machines at no extra cost. However, the SoFT model itself is not tolerant to hardware faults. In fact, faults in the hardware of the node where the set of components comprised by the controller, the assessment components, and the error handling modules are located can shut down the
whole SoFT module. Hence, these components should be replicated. The replicas, however, do not need to be diversely designed, since they are supposed to have an acceptable degree of reliability. From the viewpoint of the operating system, these are ordinary components and, as such, they can be made tolerant to hardware faults by means of well-known fault-tolerance techniques based on identical replicas (section 2.3).

The necessary number of replicas of the controller, of the assessment components, and of the error handling modules varies according to the application. For example, in a configuration based solely on conformity test, there must be as many replicas of these components as the number of versions, if hardware failures are not to interfere with the correct behaviour of the SoFT module. For an application based on majority voting, however, it suffices to ensure that there will be at least one operational voter when more than half of the versions is still executing. Hence, \(\left\lceil \frac{n+1}{2} \right\rceil\) is the necessary and sufficient number of replicas for SoFT modules based on majority voting.

6.5 Testing

The claim that the cost of developing several dissimilar solutions of the same problem is higher than making a single version is usually true. However, the cost of testing diversely designed versions is not a linear function of the number of versions. Designers can exploit the existence of dissimilar versions and, instead of executing each of the versions under controlled inputs whose outputs are known in advance, utilise random inputs and compare the result of each of the versions with the outputs of the remaining versions [KMM90] to check against discrepancies. This practice allows for a more comprehensive testing in less time. In addition, it has been demonstrated [BKA90] that, provided that most of the faults are not shared by all the versions, software testing using multiple versions is almost as effective as ideal checking, which uses an oracle for assessment. Although coincident faults can
lead to undetected errors in one particular test, it will remain undetected only if it produces similar errors for all the test cases.

The flexible structure of the SoFT model allows for the use of a particular configuration during the testing phase and a different one during the operation phase, without altering the design of the components. Since the versions can be designed independently from the fault-tolerance technique that will be adopted, the technique used in both phases can vary. In the testing phase, less efficient, none-the-less more observable, assessment components can be used. Analogously, the assessment components themselves can be tested by configuring them into a test SoFT module whose versions are especially designed to allow artificial error injection.

The replacement of a module by a new release involves the risk of the new version containing faults that were not present in the previous one. Even if the new version has not been designed to execute in a fault-tolerant configuration, it can be configured in a recovery-block-like SoFT module, so that the old version can take over if an error is detected in the new one. This configuration can be maintained until the error data collected is sufficient to ensure that the degree of reliability of the new release is acceptable. This type of configuration prevents the possibly less efficient old version from reducing the performance of the new version. In the testing phase, however, as performance is not a relevant issue, the two versions can be configured in an N-version layout, so that the outputs of the new and the old versions are always compared with each other.

6.6 Summary

The illustration of producer, consumer, client, and server modules in the SoFT framework is a topic which was also studied in this chapter. In addition, it was shown the implementation of four well-known software fault-tolerance techniques in the SoFT framework, namely: compensation, exception handling, recovery block,
and N-version programming. Although one may decide to implement SoFT modules based uniquely on one of these techniques, there are applications in which it might be necessary to mix or augment these techniques.

The analysis of the messages exchanged in interactions with the SoFT framework shows that their complexity grows linearly (i.e., is $O(n)$) with the number of versions. In some configurations, namely those in which the versions play the role of servers, execute sequentially and have their outputs assessed solely by the conformity test, the complexity of messages is $O(1)$.

The single point of failure composed of the controller, assessment components, and error handling modules has been discussed. It has been shown that the necessary degree of availability of these components could be achieved by well-known fault-tolerance techniques based on identical replicas (section 2.3).

Finally, the flexibility of the SoFT model as a framework for testing both versions and assessment components was commented.
Chapter 7

Assessment of the SoFT Controller

Modularity is a key issue for building composite components from reusable modules. The SoFT model is based on modular components which can be combined in a number of different ways. The design of the controller, which co-ordinates the other components of the model, does not depend on their internal logics or on the syntax or on the semantics of the messages that they exchange indirectly amongst one another via the controller. This independence suggests that the necessary number of different controllers is very reduced.

In this chapter, the modular approach of the SoFT model is justified. In addition, the valid configurations of the SoFT model are enumerated. They will be used in the analysis of the generic SoFT controller, which has been designed and implemented as part of the research. The aim of this analysis is to test the hypothesis that a small number of controllers suffice to co-ordinate all valid configurations of the components of the SoFT model.
Chapter 7. Assessment of the SoFT Controller

7.1 Modular Structure

The most important architectural contribution of the SoFT model is the encapsulation the different functions of fault-tolerant software modules into separate, reusable components. Whilst it seems natural to implement each diversely designed version in a separate component, it is not obvious that the remaining functions of a fault-tolerance framework, known as fault-tolerance functions, namely assessment, error handling, and the control of their interactions with one another and with the versions, should be uncoupled. This section explains the convenience of adopting a modular structure such as that of the SoFT framework, instead of building fault-tolerant software modules based on monolithic fault-tolerance components, such as the one depicted in figure 7.1.

Monolithic fault-tolerance components, such as component $FT$ in figure 7.1, may need to be redesigned when one of its functions changes, for example, when a majority voter is to be replaced by a conformity test. Monolithic fault-tolerance components could need to be redesigned even if the error handling procedures were the only necessary modification. Hence, a modular structure is more suitable than a monolithic one.
Chapter 7. Assessment of the SoFT Controller

The fault-tolerance functions in the SoFT model are realised as separate components which maintain their interfaces, despite possible changes in the remaining components. The separation of the fault-tolerance components into different modules has another major advantage, which is to cater for the replacement of inadequate components by other library components, instead of newly designed ones. Although this may not be always feasible, it is possible to create a number of components, such as a majority voter, or a conformity test which checks whether an output is sorted in a particular order, which can be used in a number of different applications.

In practical experiments in this research, based on the programming environment described in appendix C, fault-tolerance components have been implemented as independent modules that execute concurrently on a multi-task platform. However, in other programming environments, each set of fault-tolerance components, namely assessment components, error handling modules, and controller, could be implemented within a single sequential process, provided that they encapsulate data and have well defined interfaces. In object-oriented systems, for instance, they could be implemented as a set of sequential objects. Should the versions communicate by more than one port, each set of fault-tolerance components could still be configured as sequential objects, but the sets would execute concurrently.

Experience gained with practical implementations of the controller and other components has shown that the control function is more complex than some assessment functions, such as majority voting and a conformity test which assures a particular ordering. For example, whilst a majority voter has been programmed with 96 effective lines of code and a conformity test that verifies whether the messages sent by the versions are in alphabetical order has been implemented with 30 effective lines of code, the implementation of the controller has required 894 effective lines of code: approximately 9 times more than the voter and 30 times more than the conformity test. Although the complexity of a piece of code cannot be measured in terms of its size or number of lines of code, a program significantly larger than another is also likely to be more complex. The counting of effective lines of code
has included only lines with statements, loop block start declarations, and function calls. Variable declarations, end of block declarations, and return instructions have not been counted.

The controller has been designed to be generic, i.e., to be capable of dealing with any possible configuration of versions, assessment components, error handling modules, and compensation modules. Hence, for any given configuration, there are always parts of its code that are not invoked. An alternative approach is to utilise specialised, thus smaller controllers. Specialised controllers are capable of dealing only with particular configurations of versions, assessment components, error handling modules, and compensation modules. Analysis of the code of the generic controller shows that even if specialised controllers were implemented for each possible configuration, they would still be larger than the two examples of assessment components cited in the previous paragraph: at least 2.4 times larger than the voter and 7.8 times larger than the conformity test (table 7.3).

7.2 Implementation of the Controller

There are thirty-five possible combinations of the role played by a SoFT module (consumer, producer, client, and server), the execution mode of the versions (sequential, and concurrent), the order in which output messages are to be assessed (static, and dynamic), and the existence or absence of input validation. Twenty-two of them are illustrated in figure 7.2. Where applicable, the configuration caters for input validation. The sixteen configurations that are not shown in figure 7.2 differ from configurations a, and h to v by not supporting input validation. Since the size of the controller code responsible for dealing with input validation (eleven lines long) is negligible compared to the size of the complete code, it has been assumed, for the sake of argument, that configurations which cater for input validation are approximately the same size as those which do not. N/A in configuration a stands for not applicable, because consumer modules do not produce outputs.
Chapter 7. Assessment of the SoFT Controller

The maximum number of controllers that are needed in any SoFT module is thirty-five. Configuration parameters would still be needed at each of these controllers in order to specify the number of versions and error handling modules present in the fault-tolerant module, and the maximum waiting times (time-out values) to be used in communications with the other components of the fault-tolerant module.

Notwithstanding the differences that were bound to exist amongst the controllers of the different configurations, preliminary studies suggested that most of the routines would be utilised in more than one configuration. In the absence of experimental data that indicated the size of the code shared by controllers of different configurations, it has been assumed the hypothesis that a single, generic controller could be designed and implemented. This generic controller identifies the configuration of the SoFT module by means of configuration parameters and primitive return codes.

Configuration parameters inform the controller of:

![Figure 7.2: Possible Configurations of SoFT Components](image-url)

The maximum number of controllers that are needed in any SoFT module is thirty-five. Configuration parameters would still be needed at each of these controllers in order to specify the number of versions and error handling modules present in the fault-tolerant module, and the maximum waiting times (time-out values) to be used in communications with the other components of the fault-tolerant module.

Notwithstanding the differences that were bound to exist amongst the controllers of the different configurations, preliminary studies suggested that most of the routines would be utilised in more than one configuration. In the absence of experimental data that indicated the size of the code shared by controllers of different configurations, it has been assumed the hypothesis that a single, generic controller could be designed and implemented. This generic controller identifies the configuration of the SoFT module by means of configuration parameters and primitive return codes.

Configuration parameters inform the controller of:
Chapter 7. Assessment of the SoFT Controller

- the number of versions present in the fault-tolerant module
- the number of error handling modules present in the fault-tolerant module
- the execution mode of the versions
- the assessment mode of the version outputs
- if the voter precedes the conformity test or vice-versa, in configurations in which both assessment components exist
- the type of the port at which it communicates with external modules
- the maximum waiting times (time-out values) to be used in communications with the other components of the fault-tolerant module.

Some of these parameters, such as the order of precedence between the voter and the conformity test, time-out values, the number of versions, and the number of error handling modules can be changed by re-configuration messages without terminating the controller. The controller assumes that there are as many compensation modules as versions. Thus, for each port that it creates to communicate with a version, it creates another one to send messages to its associated compensation module. In addition, it automatically creates a port to communicate with the input validation module, another one for interacting with the voter, and a third one for exchanging messages with the conformity test, even if not all of them are present. The controller ascertains the assessment components which have been configured from the return code of communication primitives.

7.3 Numeric Analysis

The numeric analysis that follows offers an insight of the SoFT controller implemented in this research. It provides experimental data that shows that, although a single, generic controller is realisable, it is preferable to have a small number of
semi-specialised controllers, each of which is to be utilised at a particular class of configurations of the SoFT model.

The generic controller developed in this research contains fifty-two subroutines. Table 7.1 lists their names alongside their sizes in terms of effective lines of code. The last two columns indicate the hypothetical size of the functions which would be different if the controller were specialised in a single configuration and if no re-configuration option were available. Function reconfigure is invoked whenever the controller receives a re-configuration message. Thus, it would not exist in controllers that do not support on-line re-configuration. Other subroutines which deal with re-configuration messages would be reduced in size. The sizes of function controller is shown as zero in the last two columns for a different reason. The sole purpose of function controller is to invoke the appropriate subroutine according to the configuration of the SoFT modules. Hence, in a controller specialised in a single configuration, such a function would not exist.

The utilisation of the subroutines in the various configurations of the SoFT controller is shown in table 7.2. The subroutines are listed in the same order as they appear in the controller code. From table 7.2, it is clear that most of the subroutines are invoked in most configurations. The higher the index of the subroutine, the less utilised it is. This is so because subroutines at the bottom of the table are more specialised than those at the top. Function controller is an exception to this rule. It is the subroutine which is responsible for deciding the configuration of the controller. In addition, one can easily identify from table 7.2 the subroutines which would not exist in a particular specialised controller.

The hypothetical size of each specialised controller, which can be derived from tables 7.1 and 7.2, are stated in table 7.3. It can be verified that the specialised controllers would share between 37% (consumer SoFT modules) and 68% (client SoFT modules where the versions execute concurrently, and have their outputs assessed in a dynamic order, either by a conformity test alone or by a voter followed
## Table 7.1: SoFT Controller Function Sizes

<table>
<thead>
<tr>
<th>f</th>
<th>name</th>
<th>total size</th>
<th>single conf. size</th>
<th>no re-conf. size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>set_int_frags_max_sizes</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>set_max_sizes</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ascii_to_unsigned</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>is_number</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>unsigned_to_ascii</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>free_com_buffer</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>alloc_com_buffer</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>copy_com_buffer</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>duplicate_com_buffer</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>alloc_int_buffers</td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>receive_res</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>request_complete</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>set_timer</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>read_message</td>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>receive_complete</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>alloc_output_status_list</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>alloc_csl_and_al</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>free_extended_buffer</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>free_all_extended_buffers</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>set_version_port_name</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>set_cm_port_name</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>set_ehm_port_name</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>successful_communication</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>input_validation</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>error</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>export_to_a_single_version</td>
<td>13</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>27</td>
<td>export_to_version</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>send_to_external</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>send_to_cm</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>send_past_result</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>send_n_versions_to_voter</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>vote</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>conformity_test</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>add_version</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>log_past_message</td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>check_missing_version</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>new_message</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>ext_message_trigger</td>
<td>23</td>
<td>20</td>
<td>17</td>
</tr>
<tr>
<td>39</td>
<td>int_message_trigger</td>
<td>40</td>
<td>29</td>
<td>26</td>
</tr>
<tr>
<td>40</td>
<td>further_version_messages</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>more_version_message</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>send_and_log</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>initiate</td>
<td>109</td>
<td>88</td>
<td>87</td>
</tr>
<tr>
<td>44</td>
<td>reconfigure</td>
<td>92</td>
<td>92</td>
<td>0</td>
</tr>
<tr>
<td>45</td>
<td>concurrent_incoming</td>
<td>28</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>46</td>
<td>sequential_outgoing</td>
<td>44</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>47</td>
<td>concurrent_outgoing</td>
<td>33</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>48</td>
<td>rin_seq_cntr</td>
<td>49</td>
<td>29</td>
<td>25</td>
</tr>
<tr>
<td>49</td>
<td>rout_seq_cntr</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>rin_conc_cntr</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>rout_conc_cntr</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>controller</td>
<td>31</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### Table 7.2: SoFT Controller Function Utilisation

| f | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v |
| 1 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 2 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 3 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 4 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 5 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 6 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 7 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 8 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 9 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 10 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 11 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 12 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 13 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 14 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 15 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 16 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 17 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 18 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 19 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 20 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 21 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 22 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 23 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 24 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 25 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 26 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 27 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 28 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 29 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 30 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 31 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 32 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 33 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 34 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 35 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 36 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 37 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 38 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 39 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 40 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 41 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 42 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 43 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 44 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 45 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 46 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 47 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 48 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 49 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 50 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 51 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
| 52 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |

Chapter 7. Assessment of the SoFT Controller
Chapter 7. Assessment of the SoFT Controller

Table 7.3: Sizes of the Possible Configurations of the SoFT Controller

<table>
<thead>
<tr>
<th>Configuration</th>
<th>total size</th>
<th>size with no re-configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>327 (37%)</td>
<td>217 (28%)</td>
</tr>
<tr>
<td>b</td>
<td>534 (60%)</td>
<td>424 (54%)</td>
</tr>
<tr>
<td>c</td>
<td>525 (59%)</td>
<td>415 (53%)</td>
</tr>
<tr>
<td>d</td>
<td>534 (60%)</td>
<td>424 (54%)</td>
</tr>
<tr>
<td>e</td>
<td>575 (64%)</td>
<td>465 (59%)</td>
</tr>
<tr>
<td>f</td>
<td>566 (63%)</td>
<td>456 (58%)</td>
</tr>
<tr>
<td>g</td>
<td>575 (64%)</td>
<td>465 (59%)</td>
</tr>
<tr>
<td>h</td>
<td>568 (64%)</td>
<td>458 (58%)</td>
</tr>
<tr>
<td>i</td>
<td>559 (63%)</td>
<td>449 (67%)</td>
</tr>
<tr>
<td>j</td>
<td>568 (64%)</td>
<td>458 (58%)</td>
</tr>
<tr>
<td>k</td>
<td>609 (68%)</td>
<td>499 (64%)</td>
</tr>
<tr>
<td>l</td>
<td>600 (67%)</td>
<td>490 (63%)</td>
</tr>
<tr>
<td>m</td>
<td>609 (68%)</td>
<td>499 (64%)</td>
</tr>
<tr>
<td>n</td>
<td>399 (45%)</td>
<td>289 (37%)</td>
</tr>
<tr>
<td>o</td>
<td>417 (47%)</td>
<td>307 (39%)</td>
</tr>
<tr>
<td>p</td>
<td>426 (48%)</td>
<td>316 (40%)</td>
</tr>
<tr>
<td>q</td>
<td>489 (55%)</td>
<td>379 (48%)</td>
</tr>
<tr>
<td>r</td>
<td>507 (57%)</td>
<td>397 (51%)</td>
</tr>
<tr>
<td>s</td>
<td>516 (58%)</td>
<td>406 (52%)</td>
</tr>
<tr>
<td>t</td>
<td>530 (59%)</td>
<td>420 (54%)</td>
</tr>
<tr>
<td>u</td>
<td>548 (61%)</td>
<td>438 (56%)</td>
</tr>
<tr>
<td>v</td>
<td>557 (62%)</td>
<td>447 (57%)</td>
</tr>
<tr>
<td>generic</td>
<td>891 (100%)</td>
<td>781 (100%)</td>
</tr>
</tbody>
</table>

by a conformity test) lines of code with the generic controller. Since a significative number of lines of code (110) are necessary to cater for on-line re-configuration, these fractions are smaller when no re-configuration code is included.

These results reject the hypothesis that, although realisable, a single, generic controller would be the best approach towards the realisation of the SoFT controller. Residual faults in unused subroutines would not cause errors, because these subroutines would never be invoked. However, the figures depicted in table 7.3 are not big enough to justify the instantiation of a single, generic controller from a software engineering point of view. This is even more relevant when the fault-tolerant module communicates via multiple ports, because, in this case, there will be a controller for each set of equivalent version ports.
Chapter 7. Assessment of the SoFT Controller

153

## Class Assessment Strategy

<table>
<thead>
<tr>
<th>Role</th>
<th>Execution Mode</th>
<th>Output Assessment Indices</th>
<th>Assessment Strategy</th>
<th>Voter</th>
<th>Conf. Test</th>
<th>Voter + Conf. Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer</td>
<td>Concurrent</td>
<td>N/A</td>
<td>(a)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Producer</td>
<td>Concurrent</td>
<td>Static</td>
<td>(b, c &amp; d)</td>
<td>100%</td>
<td>98%</td>
<td>100%</td>
</tr>
<tr>
<td>Producer</td>
<td>Concurrent</td>
<td>Dynamic</td>
<td>(e, f &amp; g)</td>
<td>100%</td>
<td>98%</td>
<td>100%</td>
</tr>
<tr>
<td>Client</td>
<td>Concurrent</td>
<td>Static</td>
<td>(h, i &amp; j)</td>
<td>100%</td>
<td>98%</td>
<td>100%</td>
</tr>
<tr>
<td>Client</td>
<td>Concurrent</td>
<td>Dynamic</td>
<td>(k, l &amp; m)</td>
<td>100%</td>
<td>99%</td>
<td>100%</td>
</tr>
<tr>
<td>Server</td>
<td>Sequential</td>
<td>Static</td>
<td>(n, o &amp; p)</td>
<td>94%</td>
<td>98%</td>
<td>100%</td>
</tr>
<tr>
<td>Server</td>
<td>Concurrent</td>
<td>Static</td>
<td>(q, r &amp; s)</td>
<td>95%</td>
<td>98%</td>
<td>100%</td>
</tr>
<tr>
<td>Server</td>
<td>Concurrent</td>
<td>Dynamic</td>
<td>(t, u &amp; v)</td>
<td>95%</td>
<td>98%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 7.4: Specific SoFT Controllers Grouped by Role and Version Execution Mode

The invalidation of the generic controller hypothesis, however, does not negate the benefit of a small set of controllers, instead of thirty-five of them. From the figures shown in table 7.3, one can clearly see that the output assessment strategy does not have a great influence on the size of the controller code, but the other variables (namely: the role played by the controller, the mode in which the versions execute, and the order in which the outputs of the versions are sent to the assessment components) do. This suggests that a set of eight (the number of valid combinations of these three variables) semi-specialised controllers, which allow only the assessment strategies to vary, would have a very small amount of code which would never be used at a particular configuration. The data shown in table 7.4 corroborate this assumption. It can be seen that in the worst case (a server in which the versions execute sequentially with their outputs assessed in a static order by a conformity test only), merely 6% of the code of the semi-specialised controller will not be invoked, and in the best cases, 100% of the code will be used. These semi-specialised controllers would still permit on-line change of the assessment strategy.

The need for more comprehensive on-line changes in the configuration of the controller would lead to an even smaller set of different, less specialised controllers. However, they would have a greater proportion of unused code at each particular configuration. For example, only five controllers would be necessary, if the output assessment order were also allowed to vary, but the percentage of shared lines of
code would be reduced to the range of 81% (for servers where the versions executed concurrently) to 93% (in various other configurations). Thus, one can conclude that there is no absolute optimum number of controllers. The number of different controllers that ought to be created is essentially a technical decision that has to take into account not only reliability, but also engineering and re-configuration aspects.

7.4 Summary

This chapter explained the reasons for the separation of assessment components, error handling modules, and co-ordination functions into independent components with well-defined interfaces. This was followed by a numeric analysis of the generic controller that has been designed and implemented as part of the research. From this numeric analysis, it was concluded that the hypothesis of realising SoFT controllers as instances of a single, generic controller did not hold on the grounds that a large proportion of the generic code would not be used by controllers that deal with a single configuration of the SoFT model.

The implementation of a controller which is capable of dealing with all valid configurations of the SoFT model demonstrated the feasibility of smaller, more specialised controllers. Furthermore, it has provided valuable experimental data for estimating the size of SoFT controllers, and for balancing unused code against on-line re-configuration requirements.
Chapter 8

An Application in Air Traffic Control

SoFT has been conceived with the purpose of providing designers with a comprehensive and flexible framework for creating fault-tolerant software modules from reusable components. This hypothesis has been tested in an experiment in Air Traffic Control which involved a group of second year undergraduate students in Computer Science at University College London. The area of Air Traffic Control was chosen for the disastrous consequences that software faults could have, particularly in terms of loss of human lives.

The experiment consisted of developing a fault-tolerant software system to aid air traffic controllers on a series of tasks. Some of the modules were diversely designed by different students. These versions were then used as components of software fault-tolerant modules based on the SoFT framework.

8.1 Objectives

The air traffic control system described in this chapter aimed to test the applicability of the SoFT model by independent teams of designers in a realistic application. It would be regarded as successful even if fundamental assumptions of the SoFT...
model were rejected, because it would identify the points in which the model would have to be modified. However, it was expected that confidence in the SoFT model would increase as a result of the experiment. In particular, the experiment should demonstrate that:

- software components developed to execute as single-version modules can be configured as versions of SoFT modules without changing their original design and implementation;

- modules can be implemented either as single-version components or as SoFT modules without it being necessary to change the interfaces of the modules that communicate with them;

- the assessment components supported by the SoFT model are adequate and sufficient to prevent erroneous messages produced by the versions contaminating other modules in the system;

- programming paradigms based on data encapsulation and indirect addressing are appropriate to design, implement, and test independently the building blocks of SoFT modules.

The experiment has not set out to ascertain the effectiveness of fault-tolerance techniques as a means of improving the degree of reliability of software systems. This has already been done elsewhere [AL88, ABHM88] and is therefore out of the scope of this research. Therefore, the aim of the experiment was to identify the difficulties in building a fault-tolerant software system based on the SoFT framework, rather than to compare the reliability of the fault-tolerant modules with the reliability of their respective single-version configurations.
Chapter 8. An Application in Air Traffic Control

8.2 The Experiment

Six second year undergraduate Computer Science students at University College London were invited to design and implement a software system to aid air traffic controllers on a series of tasks. (The principles of air traffic control are briefly described in appendix D.)

8.2.1 Executive Overview

The students were divided into two groups. They were formally notified not to discuss their work with members of the other group. Each group played the role of a software house which was invited by a contracting company (the students' supervisors) to produce software modules for an air traffic control system, ATCS in short. The remaining modules were produced by the supervisors, due to the limited amount of time that the students had to dedicate to the project.

The project was divided into three stages: specification, development, and documentation. In the specification phase, the two groups had joint meetings with the supervisors in order to define the software modules and their interfaces. No comment was made as to the internal design and implementation of the modules. During the implementation phase each group had private meetings with the supervisors. When the implementation was completed, the two groups merged in order to produce the documentation [HSD+93].

One of the requirements of the project was for it to be possible to produce single-version systems from the modules developed by each of the groups, complemented by the modules created by the supervisors. Furthermore, it should be feasible to replace any component designed by the students by its redundant version without altering the design of the system. The modules that were designed and implemented by both groups were then used as diversely designed versions of SoFT modules in
Chapter 8. An Application in Air Traffic Control

8.2.2 Technical Description

The students designed and implemented a series of software modules that performed the following tasks: validation of flight plans, extraction of flight route, estimation of report times, aircraft collision warning, and scheduling of arriving aircraft. Since the first two activities are sequential and can be executed only once for each flight plan that is activated, they were grouped in the same module. The remaining three were implemented as separate, concurrent components.

The modules developed by the students were complemented by four sub-systems implemented by the supervisors. The first of these sub-systems provides the interface with the air traffic controller. The second simulates radar data. The third one supplies the current time on request and sends a void message at regular time intervals (every five minutes in the current implementation). This is to be used to trigger periodical events in other modules. Flight plans are stored in files which are created off-line by a fourth program.

Figure 8.1 depicts the general layout of the ATCS. Module Plan is responsible for the validation of flight plans and for the extraction of the flight route. It receives a flight plan from module AT Controller, which has read it from a file input off-line by InPlan. When the syntactical analysis of the flight plan is completed, Plan responds to AT Controller via port new indicating whether or not the plan is syntactically correct. The flight route is then extracted from the route field of valid flight plans. The names of the air routes and report points were taken from the air route chart of the United Kingdom, depicted in figure 8.2. The route is then sent to module Estimate for the calculation of the report times.

Faults in module Plan may lead to the rejection of a correct flight plan. The flight plan is, thus, terminated. In order for the flight plan to be activated despite
Figure 8.1: Diagram of the Air Traffic Control System
its rejection by module \textit{Plan}, the air traffic controller is entitled to bypass the syntactical analysis phase. This phase is also bypassed when repetitive flight plans (section D.1.1) are activated, with the exception of when they are activated for the first time or when they are modified.

When a new flight route is received by module \textit{Estimate}, it produces and sends to \textit{AT Controller} a list containing the names or the geographical co-ordinates of the points and the estimated times at which the pilot will establish radio contact with the air traffic controller. When the pilot reports the aircraft position, the air traffic controller inputs this information into the computing system via module \textit{AT Controller}. As a result, \textit{AT Controller} sends a message to \textit{Estimate} via port \textit{rep} stating that an aircraft has reported at a particular point. \textit{Estimate} then reads the current time and, if the error between the estimated and the actual report times is greater than a pre-established value (three minutes, in this example), it recalculates the remaining estimated report times and sends them to \textit{AT Controller}. In addition, if the flight is to be transferred to an approach control office (indicating that it will land) module \textit{Estimate} transmits the identifications of the flight and of the aerodrome to component \textit{Scheduler}.

Module \textit{Scheduler} provides the air traffic controller with a list of flights which are scheduled to land at a particular aerodrome in the near future and a list of flights which, although flying in the vicinity of the aerodrome, will not land there. \textit{Scheduler} is triggered periodically (every five minutes, in the current implementation) in order to inspect the list of aircraft within radar range. This message is generated by component \textit{Clock}. \textit{Scheduler} selects the aircraft within a pre-defined distance from the aerodrome (70 nautical miles, in the present case) to create the two lists. Flights that will not land at the aerodrome are listed in a random order. The sequence in which flights scheduled to land are enumerated minimises the total time elapsed between the first and the last aircraft touching the runway.

Module \textit{Radar} simulates the data produced by the combination of the primary
and secondary surveillance radars. It periodically sends out messages containing
the flight identification, the location of the aircraft expressed in geographical co­
oordinates, and its altitude. The first two data are used by module *Scheduler* to
ascertain which aircraft are within a pre-defined distance from an aerodrome. Mod­
ule *Collision* utilises the complete information in order to verify whether any two or
more aircraft have violated the separation minima (section D.1.2). Should this hap­
pen, *Collision* will send a warning message to *AT Controller* with the identification
of the aircraft which are on collision routes.

### 8.3 Incorporation of Software Fault-Tolerance

Modules *Plan*, *Collision*, and *Scheduler* have been replicated. Each of these modules
can be configured either as a single-version component or as a software fault-tolerant
component, where the two dissimilar replicas are used as SoFT versions. Originally,
*Estimate* was to be replicated, but, due to the limited time that the students had to
complete the project, only one version of this module was produced. The different
configurations of the SoFT modules in the ATCS are reported and discussed in this
section.

#### 8.3.1 Plan Validation

Module *Plan* indicates whether or not the syntax of a flight plan is correct by means
of an ASCII message. The only valid messages are: *OK*, and *Not_OK*. A flight plan
is approved only if *both* versions agree that the plan is correct. Otherwise, the air
traffic controller is advised to examine it manually.

In the fault-tolerant configuration of module *Plan*, the assessment of its reply
to *AT Controller* is done by an *absolute voter*\(^1\). In this particular configuration, in

\(^1\) *Absolute voting* requires that *all* versions agree on the same output.
which only two versions exist, majority voters and absolute voters are interchangeable. However, when three or more versions are configured, confidence in the outputs which have been accepted is increased by using an absolute voter. The absolute voter prevents the acceptance of an incorrect output which result from failure modes which are common to the majority of the versions. The SoFT controller is configured to send the outputs of the versions for assessment in a first-come-first-served order, because the outputs of all the versions are needed and there is no priority between them.

It must be emphasised that the absolute voter is acceptable in this configuration only because there is an operator who can override the verdict of the absolute voter (section 8.2.2), should he/she find the plan to be valid, and because the number of occasions in which the versions disagree is expected to be very small. Absolute voters are not appropriate in unmanned systems in which an output produced by the majority of the versions is better than none at all.

The versions will normally agree on the output, be it an OK or a Not_OK message. An error will have occurred if the versions disagree. In these cases, the voter commands the SoFT controller to call an error handling module which replies with a Not_OK message to be forwarded to module AT Controller via port new. In addition, the error handling module sends an error message to module AT Controller via port error, so that the error be logged for off-line analysis.

Common failure modes result in coincident incorrect outputs, which will not be detected by the voter. It is impossible to detect common errors that result in a valid message, namely OK and Not_OK. However, invalid messages may be discarded by a conformity test. Thus, the voting is complemented by a conformity test. Should the outcome of the voting, which is submitted to the conformity test, be an invalid message, the same error handling module that is to be invoked by the voter will be called.
The same error handling module is invoked by the SoFT controller when a version fails to respond. Hence, it is configured as the universal error handling module. The SoFT configuration that produces the reliable reply to module \textit{AT Controller} is depicted in the left hand side of the diagram of figure 8.3.

### 8.3.2 Route Extraction

Module \textit{Plan} sends a message to module \textit{Estimate} via port \textit{route} whenever a flight plan is activated. This message contains a list of the aircraft report points with their respective cruise levels and speeds. In the fault-tolerant configuration of module \textit{Plan}, this message is generated by a bit-by-bit comparison of the outputs of the versions. The assessment is complemented by a conformity test which detects common representation errors. This configuration is illustrated on the right hand side of the diagram of figure 8.3. It is another example in which it is advantageous to
combine the two output assessment components supported by the SoFT framework.

Should the assessment components fail to agree upon a reliable message, an error handling module is invoked. The error handling module sends the error message to AT Controller via composite component port error.

SoFT module Plan contains two SoFT controllers: one linked to port new and another to port route. The two controllers do not need to synchronise, because only the controller linked to port new sends messages to the versions (section 5.6.1).

### 8.3.3 Approaching Lists

Module Scheduler produces two types of lists of flight identifications. One, which is written to port sched, contains the identifications of the flights that are due to land at a particular aerodrome. The other, which is written to port unsch, informs the air traffic controller of the flights that are in the vicinity of a particular aerodrome but will not land there. In both lists, the worst possible error is to omit the identification of a flight that ought to be in the list. Therefore, two identical SoFT configurations are employed to produce reliable outputs from the messages produced by the versions. In both configurations, the assessment is made by a voter which issues a message formed by the union the individual messages sent by the versions. These configurations are illustrated in the diagram of figure 8.4 by controller $C$ and voter $V_1$, and by controller $C'$ and voter $V'_1$, respectively. In order to simplify the diagram, only ports which are linked have been represented in the SoFT controllers of figure 8.4.

The main task of module Scheduler is to supply the air traffic controller with the flights scheduled to land, sorted in an optimal order in terms of the total landing time of all aircraft. Thus, the air traffic controller must be advised if the list received by module AT Controller at port sched does not meet this requirement. Another SoFT controller, $C''$, is linked to the output of SoFT controller $C$ in order to verify
Chapter 8. An Application in Air Traffic Control

Figure 8.4: Fault-Tolerant Configuration of Module Scheduler
whether the list is optimal or not. The verification is done by a conformity test ($CT$). The output port of controller $C''$ is not linked, because, if the list is found to be optimal, no warning must be issued. If $CT''$ rejects the list, an error handling module ($EHM''$) is invoked. As a result, an error message is written to port $error$. Even when the outcome of voter $V$ does not lead to the minimum overall landing time, the air traffic controller should be made aware of the flights that are preparing to land at the aerodrome. Hence, $CT''$ could not have been combined with voter $V$ in the same SoFT configuration, because it would prevent SoFT controller $C$ from exporting non-optimal lists.

The same flight identification may appear in both lists, if a byzantine error occurs. This erroneous behaviour is detected by a fourth SoFT configuration which votes the outputs of SoFT controllers $C$ and $C'$. The assessment of the fourth SoFT configuration is based on a voter ($V_v$) whose final output is the intersection of the outputs of SoFT controllers $C$ and $C'$. The output port of the fourth SoFT controller ($C'''$) is linked to composite component port $error$. No message is sent to $AT Controller$ if the intersection of the two lists is null.

The two versions of $Scheduler$ import trigger messages from port $timer$ and position messages from port $pos$. No input validation procedure is used in the fault-tolerant module $Scheduler$. However, composite component ports $timer$ and $pos$ are not linked directly to the version ports with the same names, because concurrency and mainly distribution could lead to both versions being triggered when only one of them had received the last position message. Two SoFT controllers, which synchronise by means of the protocol described in section 5.6.1, are configured to ensure that both versions read trigger and position messages in the same order. To simplify the diagram, these SoFT controllers are not represented in figure 8.4.

8.3.4 Collision Warning

$Collision$ is the most critical module of the ATCS, because it involves the safety
Chapter 8. An Application in Air Traffic Control

Figure 8.5: Fault-Tolerant Configuration of Module Collision

of human lives. Its versions may fail either by reporting aircraft that are not on collision course or by omitting the identification of aircraft that have violated the separation minima. Since it is preferable to raise a false alarm than to fail to detect a dangerous situation, the output of the fault-tolerant Collision module is the result of the union of the two sets of aircraft identifications produced by each of the dissimilar versions. No conformity test is used in combination with the voter. Modules $C$ and $V_0$ in the diagram of figure 8.5 depict the SoFT controller and the voter, respectively.

In figure 8.5, the only ports of the SoFT controllers that have been illustrated are those which are linked.

As in the fault-tolerant configuration of module Scheduler, a second SoFT configuration is used to warn the air traffic controller of the differences between the messages produced by the two versions. The second SoFT configuration is based on a voter which generates the complement of the intersection of the two messages. This configuration is represented by modules $C'$ and $V_{cc}$ in the diagram of figure 8.5.
The second SoFT controller writes the outcome of voter $V_{\pi n}$ at port error.

### 8.4 Conclusions of the Experiment

The experiment described in this chapter, in which software fault-tolerance was transparently attached to history-independent modules, presents evidence in support of the suitability of the structure, components, and interfaces of the SoFT model. However, it does not imply that transparency can be achieved in history-dependent modules, as can be learned from chapter 9.

Despite the provision for error report ports at modules *Plan*, *Collision*, and *Scheduler*, no error message has been defined in their single-version implementations. When they were replaced by functionally equivalent SoFT modules, a number of possible error messages appeared, mainly in module *Scheduler*. The dissimilar versions of these modules did not need to be redesigned in their fault-tolerant configurations, neither did the modules that communicate with them. However, it would have been more appropriate to alter module *AT Controller* to deal with error messages that arise only in the fault-tolerant configuration of the ATCS. This conclusion is not specific to this example. In software systems in which new error messages appear as a result of the replacement of single-version modules by fault-tolerant components, modules that communicate with fault-tolerant components may need to be modified in order to deal with error messages that do not exist in the single-version configuration of the system.

In this ATCS, there are assessment algorithms that produce reliable messages from the union of the version outputs. Hence, the reliable output may contain elements that are present in the output of only one of the versions. Majority voting could not have been employed in this particular system, because all SoFT modules contained only two versions. Thus, majority voting could, at most, have been utilised for error detection, but not for fault-tolerance. The assessment strategy
adopted is appropriate for this application, because the air traffic controller can resolve situations in which the two versions disagree. In systems where a human operator is present, to report to the operator the values which exist only in the minority of the messages may be helpful even if three or more versions exist. This approach, however, is not suitable for unmanned control systems in which values not present in the majority of the versions must be discarded.

Finally, it is prudent to acknowledge that only in real systems, with a high degree of complexity, could the SoFT model be validated. However, notwithstanding the limited size and complexity of the ATCS described in this chapter, this experiment has succeeded in showing necessary adjustments to preliminary architectures of the SoFT model, in giving valuable insights, and in increasing confidence in the practical application of the SoFT framework.

8.5 Summary

This chapter described a project executed by second year undergraduate Computer Science students at University College London in the area of Air Traffic Control. The project consisted of the elaboration of a software system to aid air traffic controllers on a series of tasks, namely: validation of flight plans, extraction of flight route, estimation of report times, aircraft collision warning, and scheduling of arriving aircraft. One of the requirements of the system was that some functions were tolerant to software faults. Thus, after formalising the problem and specifying the necessary modules and their interfaces, the students were divided into two groups. Each group designed and implemented independent versions of the main functions. These versions were then used as components of software fault-tolerant modules based on the SoFT framework. This project offered the opportunity to test the framework and the guidelines for building software fault-tolerant components developed in this thesis in a small, but realistic, application designed and implemented by an independent team.
Chapter 9

Limitations and Related Work

In this chapter, the use of the SoFT model to provide non-transparent software fault-tolerance to history-dependent versions is discussed. In addition, it is shown that there is a limit to the degree of diversity amongst components that can be used as versions of a SoFT module. Finally, the SoFT model is compared with other approaches for designing and implementing fault-tolerant software.

9.1 Non-Transparent Error Recovery

Transparent software fault-tolerance requires that versions do not make explicit reference to assessment procedures or enumerate variables to be examined. Thus, internal checkpoints cannot be set nor can internal variables be listed as members of the version internal state. As far as SoFT is concerned, versions are black boxes whose internal variables cannot be accessed. The only observable values of these modules are their outputs.

As defined in chapter 2, the external behaviour of a system (i.e., the way that it is seen by the environment) is described by a finite set of external states. In systems based on message passing, the external states of a module are determined by the messages issued by it. Each external state is represented by the contents of a single
message.

Assessing the external states of the versions suffices to prevent them from contaminating other components of the system. However, should an error be detected, it is necessary to guarantee that the error will not interfere in future computations of the version. Stateless (history-independent) modules are the trivial case in which no recovery is necessary, because their behaviour does not depend on their previous computation. Conversely, history-dependent modules need to recover from the error, if they are to resume a correct computation.

Neither the controller nor the error handling modules can correct the internal state of the versions, because none of them have access to internal variables of the versions. Such a recovery can only be done by the versions themselves. However, although transparency has to be abandoned when history-dependent versions are used, no modification is necessary in the SoFT model.

### 9.1.1 Recovery Based on the Feedback of the Reliable Output

Compensation modules have been conceived with the purpose of compensating external actions taken by the versions prior to the assessment of their computations. The controller supplies a compensation module with the verdict of the assessment components on the output issued by the version associated with it. Moreover, if the output disagrees with the outcome of the assessment, the controller adds the outcome to the compensation message.

A version which encapsulates normal computation code and error recovery code can use the compensation message to promote internal recovery (section 4.3.2). The error recovery code can change global variables accessed and updated by the normal code, because global variables are shared by both pieces of codes. A version which encapsulates the normal computation and the error recovery codes is depicted in
Figure 9.1: Normal and Error Recovery Codes in a Single Component

Figure 9.1. The normal computation code exchanges messages with the controller via port \( n \). The error recovery code receives compensation messages at port \( r \). The thin line shows that the pieces of codes are different. However, as both are encapsulated within the same module, they have access to the same global variables.

### 9.1.2 Recovery by Assessing Internal States

The reliable output agreed by the assessment components is sufficient to correct the state of history-dependent versions in which there is a one-to-one correspondence between the internal and external states. However, those in which this assumption does not hold cannot recover from an error simply by receiving the reliable output agreed upon by the assessment components; their internal states need to be assessed. Assessing internal states reduces the independence amongst the diversely designed versions. Therefore, this practice must not be followed unless it is absolutely necessary.

The internal state of a module can be defined as the set of observable internal values that affect its behaviour. The internal state of a module is assessed at checkpoints. The variables that identify its internal state must be explicitly enumerated. Analogously, the locations of the checkpoints need to be explicitly set, unless they coincide with particular events such as transmission and reception of messages, or
Chapter 9. Limitations and Related Work

Figure 9.2: External Assessment of the Internal State of a Module

kernel calls.

The SoFT model has been conceived with the purpose of attaching software fault-tolerance to application modules transparently. Hence, it does not provide any direct mechanism for assessing internal variables. However, versions which have been designed to execute in a fault-tolerant configuration can use the SoFT framework to assess their internal variables in an analogous manner to that described in section 9.1.1.

Modules based on encapsulation can have their internal variables assessed by independent modules which are seen by them as black boxes. This configuration is represented in figure 9.2. Module \( V \) is the component whose internal variables are to be assessed. It interacts with other application modules via port \( ext \). The values of the internal variables that determine its internal state are sent to be assessed externally via port \( assess \). The result of the assessment is imported from port \( result \). Module \( A \) assesses the contents of the messages received at \( \pi \) and replies at \( \rho \) with the indication of whether or not the contents of the variables are acceptable. If not, it sends the correct values along with the status code. Although not shown in the figure, assessment module \( A \) may use other pairs of ports \( \pi_i - \rho_i \) to interact with other versions of \( V \) in order to obtain a comparison pattern.

Assessment component \( A \) of figure 9.2 can be implemented as a SoFT configuration. The interface between modules \( V \) and \( A \) is exactly the same as the one used between a non-transparent version which reads its compensation message and the SoFT controller. The assessment procedure described in the previous paragraph is
based on the same algorithm which is used by the controller to generate the compensation messages. Therefore, assessment module $A$ can be realised as a SoFT module. Depending on the assessment paradigm, either the voter, or the conformity test, or both will be configured. One or more error handling modules can be defined to generate special error messages when the assessment components do not agree on a reliable state.

Figure 9.3 illustrates SoFT module $\Sigma$. Versions $V_1, \ldots, V_n$ have their outputs assessed by voter $V$ and conformity test $CT$. In addition, their internal states are assessed by composite component $A$, which is a SoFT module. The internal states of the versions are compared to each other by voter $V'$. When $V'$ reaches a consensus, $C'$ sends the compensation messages via its local ports $\rho_1, \ldots, \rho_n$. Then, each of the versions learns from its compensation message whether or not its internal state has been regarded as reliable. The versions whose internal state does not match the one agreed upon by $V'$ can use the values reported in the compensation message to correct it. Matching voter $MV'$ is necessary to assess messages that arrive after the voting has been completed and which have not been used by $V'$ in that round of voting.

9.2 Versions with Non-coincident External States

Two or more modules are said to be redundant if they provide equivalent services. The simplest method of obtaining redundant software modules is to create several instances of the same module. However, since these instances have identical code, they cannot be used to cope with design faults. Software fault-tolerance can only be achieved by employing diversely designed modules.

Diversely designed modules may have coincident internal states. This limits diversity, but may be necessary for the versions to be capable of recovering from errors (section 9.1.2). As far as error confinement is concerned, to guarantee that the
Figure 9.3: A SoFT Module Assessing the Internal States of SoFT Versions
versions have coincident external states is sufficient. However, coincident external states are not always necessary. Two or more redundant modules may convey the same information through different sets of messages. For instance, the two versions that compose the fault-tolerant configuration of module Collision in the air traffic control application described in chapter 8 have been specified to send the identification of all aircraft that have violated separation minima criteria to the air traffic controller in a single message. However, the same information would be conveyed by a version which sent a separate message for each aircraft on a collision course with another one. If one defines external meta-state as the overall external behaviour of a module which results from its navigation through a set of subsequent external states, it can be said that modules which provide equivalent services despite issuing dissimilar messages have coincident external meta-states.

One could define degree of freedom of design as the amount of freedom that the designer has to code the module. As far as the degree of freedom of design is concerned, redundant modules can be qualitatively grouped in four different classes:

- coincident code
- coincident internal states
- coincident external states
- coincident external meta-states

Redundant modules with coincident codes have a degree of freedom of design equal to zero. Modules belonging to the other three classes have increasing degrees of freedom of design.

The SoFT model has been conceived to build software fault-tolerant modules from versions with coincident external states that do not require internal error recovery. However, versions which need to execute internal error recovery (section 9.1)
can be used as components of SoFT modules at the cost of transparency being abandoned.

Software fault-tolerance can be transparently attached to versions which have coincident external states and do not require internal error recovery, because the assessment procedures can always be triggered by a well defined event, namely the transmission of a message.

As to versions with coincident external meta-states, there is no event that can automatically inform the SoFT controller that the outputs of the versions are ready to be assessed. In addition, the number of messages that each of the versions issue in a coincident external meta-state varies according to the application. Therefore, the SoFT model cannot be utilised to build software fault-tolerant modules from versions with coincident external meta-states.

9.3 Related Work

Several models for developing fault-tolerant software have been proposed in the literature [ADG+90, BDW+92, SM90, SW]. Three of the most well-known of them, namely Recovery Blocks, DEDIX, and Circus, have been briefly described in sections 2.4.3 and 2.4.5. Each of them has its limitations and drawbacks. In this section, the SoFT model is assessed in terms of the features that make it preferable to these approaches. In addition, SoFT is compared to a fourth model, FTM, which shares some of its overall principles.

9.3.1 Canonical Recovery Blocks

Recovery blocks first appeared in the form of language constructs [Ran75]. However, this software fault-tolerance technique can be implemented in many other paradigms (for example, in an exception handling approach [CR86]). The adjective \textit{canonical}
has been added to the name *recovery block* to make it explicit that the text is referring to the *architecture* proposed in [Ran75]. Recovery blocks can be implemented in the SoFT model, as shown in section 6.2.3.

In canonical implementations of recovery blocks, when an alternative fails, a recursive cache is used to restore the state of the program at the beginning of the recovery block. This is necessary because the alternatives of the recovery block share state amongst one another and with the caller. Any change in the global state of the program must be undone before the next alternative in line executes. In SoFT, the caller and the versions are separate modules with independent states. Hence, there is no need to restore the original state of an erroneous version, because the state of the calling module will change only when the controller responds to it. In fact, all history-dependent versions, including those in a hierarchical position lower than the successful one, must process the original request in order to move to equivalent states. This apparent drawback in comparison to the shared state model is not usually a serious one as far as response time is concerned, because the controller does not need to wait for the messages produced by the lower rank versions before it can issue the reliable output. Moreover, by uncoupling the states of the versions, the "alternatives" can execute in parallel, thereby reducing or even eliminating the delay caused by waiting for the next version in line to produce its output.

Canonical recovery blocks are servers. In the SoFT framework, however, it is possible to build software fault-tolerant clients and producers based on the recovery block approach. The architecture of these SoFT modules is a mirror-image of the one shown in figure 6.12. Figure 9.4 depicts a client SoFT module based on the recovery block technique. The versions always execute concurrently, since the controller cannot determine the time at which the versions will produce their outputs. The controller can be configured to read their messages either in sequence (static assessment order) or in a first-come-first-served manner (dynamic assessment order). Matching voter $MV$ is needed because the versions execute concurrently. A producer SoFT module based on the recovery block approach can be represented
Figure 9.4: Example of a SoFT Client based on the Recovery Block Approach

by a similar diagram. It differs from the one illustrated in figure 9.4 in the types of ports $\pi_1, \ldots, \pi_n$, and $\xi$.

9.3.2 DEDIX

SoFT shares some principles with DEDIX, described in section 2.4.5. The diversely designed versions in DEDIX, like those in SoFT, do not share data. In order to tolerate hardware crashes in the nodes where the versions execute, the versions must be instantiated in different nodes of the network. SoFT and DEDIX use time-out clauses to detect errors in the time domain. In both approaches, the assessment algorithms are external to the versions. Moreover, the design of the versions is not influenced by them. SoFT supports equally fault-tolerance based on N-version programming and on the recovery block scheme. Although DEDIX has been originally conceived to cater for the former, it can realise the latter with
relative ease.

SoFT and DEDIX differ in a number of points. The DEDIX code is located at an intermediate layer between the application layer and the operating system. Assuming that several application components are realised as N-version modules, there may be various versions, each of a different module, executing on the same node. All of them communicate with the local copy of DEDIX. Thus, DEDIX must be capable of dealing with several assessments simultaneously. In SoFT, there is a copy of the controller, assessment algorithms, and error handling modules for each set of equivalent version ports. This results in much simpler components, and is thus less error-prone.

DEDIX deals with hardware and software faults concomitantly by using active replication. The messages produced by the versions are assessed at all sites. In SoFT, tolerance to hardware crashes and software fault-tolerance are orthogonal issues. The assessment components, the error handling modules, and the controller are centralised elements. They are to be replicated in order to survive hardware crashes. However, as they can usually be regarded as much more reliable than the versions, the replicas do not need to be diversely designed. Active replication is one of the options for achieving hardware fault-tolerance in SoFT, but any other fault-tolerance technique based on identical replicas (section 2.3) can be used.

The time-out function in DEDIX can start immediately after the request is sent to the versions (absolute time counting), or after the majority of versions have produced their outputs (majority time counting). SoFT uses relative time counting, which is a technique that lies between those two. In this approach, the countdown starts when the first message sent by one of the versions is received by the controller. Relative time counting is preferable to absolute time counting, because it filters common causes of delay that affect all the versions (section 5.4.1). Although majority time counting prevents early replies from triggering the countdown, it leads to a deadlock when an omission error occurs in most of the versions.
Notwithstanding the transparency of the assessment algorithms both in DEDIX and in SoFT, DEDIX always requires the versions to call the Cross-Check-function explicitly. Therefore, all versions in DEDIX must be designed to execute in this fault-tolerance environment. It has been shown in section 9.1 that, in SoFT, versions which must promote internal recovery cannot ignore the fact that they operate in a fault-tolerance framework. However, those which do not need to recover from erroneous previous computations can be configured either as versions of a SoFT module or as single-version components without requiring any alteration in their design.

9.3.3 Circus

Both SoFT and Circus hide replication from the versions by implementing a communication interface which is the same for single and multiple versions. They differ in the communication paradigm. Whilst inter-module communication in SoFT is based on message exchange, Circus has opted for the remote procedure call approach. Although remote procedure call is an elegant form of providing inter-module communication, there are situations in which this paradigm cannot be applied without abandoning its synchronous character. Remote procedure call could not be used in the communication between the SoFT controller and server versions, because only when it is configured in the sequential mode can the controller wait until a version replies before sending the external request to the next one. Such a communication could not be implemented as a multicast, because the controller must receive all the replies.

Circus splits the communication between a set of $m$ clients and $n$ servers into one-to-$n$ messages from the standpoint of each of the clients and $m$-to-one messages from the point of view of each of the servers. When a replicated request arrives at a server or when a replicated reply reaches a client, they are reduced to a single message by a collator. Unlike Circus, SoFT generates a single message from the
Chapter 9. Limitations and Related Work

$m$ requests, or from the $n$ replies, logically at the sender. This enables a SoFT module to be seen as having the same functional behaviour and interface as any of its versions and prevents single-version modules that communicate with it from being modified to have collators attached to them. Moreover, since all messages that are to be received by each of the versions are assessed at the same physical node (section 5.6), the sequencing problem (section 2.4.5) is trivially resolved.

9.3.4 Fault Tolerant Module (FTM)

SoFT is based on the principle that a framework for developing fault-tolerant software modules should support well-known software fault-tolerance techniques, as well as combinations of them. This principle is shared by the Fault Tolerant Module (FTM) model [Liu91]. The FTM model is composed of three parts: the declaration of the module interfaces, the declaration of the assessment methods, and the redundant versions.

Unlike SoFT, which does not distinguish between ports at which normal messages are exchanged and ports at which error messages are sent to external modules, FTM separates the two types. Separation makes the interface clearer, but it requires external modules to comply with the same syntax, thereby making their interfaces particular to communication with FTM modules only or to other components that use the same type of interface.

Similarly to SoFT, the assessment methods of the FTM model may be voters and/or conformity tests. The versions can be grouped in distinct sets, each of which can be assessed by a different method. For instance, FTM module $M$, illustrated in figure 9.5.a, contains four versions $V_1, \ldots, V_4$, a conformity test $CT$, and two voters $V'$ and $V''$. Voter $V'$ assesses the outputs of versions $V_1$ and $V_2$, whilst the outputs of versions $V_3$ and $V_4$ are voted by $V''$. The final output of module $M$ is then decided by conformity test $CT$, which assesses the outcomes of each of the voters in series. In SoFT, such a hierarchical assessment is not possible at the same level.
Chapter 9. Limitations and Related Work

Figure 9.5: Equivalent Server Configurations in the FTM and SoFT Models

of abstraction. In order to implement the same assessment criteria, it is necessary to create a SoFT module $\Sigma$, such as the one depicted in figure 9.5.b, composed of two versions $\Pi_1$ and $\Pi_2$ and a conformity test $CT$ to assess the messages from $\Pi_1$ and $\Pi_2$. These versions are SoFT modules themselves. $\Pi_1$ contains versions $V_1$ and $V_2$, and voter $V'$. $\Pi_2$ encapsulates versions $V_3$ and $V_4$, and voter $V''$. Allowing for hierarchical assessment in the same level of abstraction would require a more complex controller.

Alongside the declaration of the normal parameters of the versions, which must be the same for all the versions, the FTM model caters for the definition of exceptions that might be raised by each of the versions. The body of each version and the
respective exception handlers are listed in the declaration of the version. FTM assumes that a version may try to recover from an auto-detected error by raising an exception. In contrast to this approach, SoFT does not make any assumption as to the internal fault-tolerance mechanism used by the versions. Versions which provide internal error detection and recovery by means of exception handling can be realised in SoFT as shown in section 6.2.2.

The major difference between the FTM and the SoFT models is not in their approaches towards software fault-tolerance, but rather in the programming paradigm on which they are based. FTM modules are pieces of code which can sometimes be shared amongst multiple callers. Hence, concurrent access to the FTM module must be protected by atomic actions. In SoFT, atomic actions are not necessary because outputs are exported only when the assessment components have completed their analysis of the version outputs.

9.4 Summary

In this chapter, it has been concluded that history-dependent versions which need to have internal error recovery, must be designed specially to execute within the SoFT framework. Hence, software fault-tolerance cannot be transparently attached to versions that retain state. In addition, the reasons for requiring the dissimilar versions to have coincident external messages have been discussed in this chapter. Finally, the SoFT model was compared with different approaches for designing and implementing fault-tolerant software in terms of their structural similarities and differences.
Chapter 10

Conclusions

The purpose of software fault-tolerance is to provide tolerance to residual design faults in the software of computing systems by means of redundant components which fulfil the same requirements, but which have been designed according to dissimilar strategies. The development of redundant software components is bound to increase the final cost of the computing system. Therefore, only components which must be highly reliable should be made tolerant to software faults. Furthermore, ideally, the implementation of a particular software module as a fault-tolerant component should not require its original design to be modified. Neither should it be necessary for other modules that communicate with it to have a special design. Fault-tolerance should, then, be selective and transparent.

In this thesis a framework for designing and implementing software fault-tolerant modules selectively and transparently was presented and discussed. This framework, called SoFT, an acronym for Software Fault-Tolerance, is based on components that encapsulate data, address one another indirectly, and communicate by message exchange. The generic configuration of the framework has been repeated in figure 10.1. This modular approach allows for the utilisation of reusable components. The importance of reusable components in software fault-tolerant modules is twofold: firstly, reusable components contribute to a reduction in the development time and cost of software fault-tolerant modules; secondly, the behaviour of such components may
be more accurately foreseen due to the existence of operational field data from previous systems. SoFT provides an abstraction for fault-tolerant modules whereby other system components view them as regular composite components.

There are a number of software fault-tolerance techniques. The key ones were examined in chapter 2. Each of them has its benefits and limitations. They may be combined in order to mask the deficiencies of one another. The SoFT framework caters for the realisation of fault-tolerant modules based on well-known software fault-tolerance techniques, such as compensation, exception handling, recovery blocks, and n-version programming, as well as on combinations of them. From a functional point of view, the advantage of providing for the realisation of these techniques under a unified framework is reinforced by the fact that most of the code that is necessary for implementing one technique is common to other schemes (chapter 7).
Chapter 10. Conclusions

10.1 The SoFT Model

The components of the SoFT model are divided into four classes: versions, assessment, error handling, and control. The versions are responsible for the functional behaviour of the fault-tolerant module. SoFT caters for the version inputs being assessed by an input validation module and their outputs being assessed by a conformity test, a voter, or a combination of both. Should the input validation module reject an input or the output assessment components fail to agree upon an acceptable output, an error handling module is invoked in order to inform other system components of the error and/or to execute emergency procedures. Erroneous external actions taken by the versions are dealt with by compensation modules associated with them. Messages that are received from the versions after an acceptable output has been exported from the SoFT module are dealt with by a matching voter. The components of the SoFT model interact under the co-ordination of a component called the controller. Thus, briefly, the components of the SoFT model are:

- versions
- assessment components
  - input validation module
  - output validation
    * conformity test
    * voter
    * matching voter (auxiliary component)
- error handling
  - error handling modules
  - compensation modules
- controller
Software faults do not appear at random, as many hardware faults do. Replicating a piece of code does not avoid the occurrence of software errors, because the same faults will exist in all replicas. Software fault-tolerance can only be achieved by creating redundant versions which follow dissimilar strategies, whilst fulfilling the same requirements. Since software faults can be introduced in all stages of the software development process (specification, design, implementation, testing, and maintenance), diversity should be employed in all of them.

External messages addressed to the fault-tolerant module are received by the controller and submitted to the input validation module, if it exists. Invalid inputs cause the controller to invoke an error handling module. Valid inputs are forwarded to the versions.

The outputs of the versions are received by the controller and sent to the output assessment components that have been configured. Analogously to the input validation, an error handling module is called if no acceptable output can be agreed upon by the output assessment components. Errors in the time domain are detected by the controller according to time-out clauses. The values of time-out clauses can be changed from their defaults either by a configuration parameter, when the controller is initiated, or by a re-configuration message.

When the output assessment has been completed, the controller issues a compensation message to each version whose output has been assessed. A compensation message contains the verdict on the output of the version and, if an error has been detected, the output agreed upon by the assessment components. The original purpose of compensation messages is to allow for the version (or a compensation module on its behalf) to amend incorrect external actions which have been executed prior to the output assessment. However, they also provide a feedback to versions specially designed to be configured in the SoFT framework. The versions can use this feedback to promote internal error recovery and to update an error log. Alternatively, the version can delay external actions until it receives the compensation message.
The number of outputs that need to be assessed in order for an acceptable output to be produced is usually smaller than the number of versions. Therefore, outputs may be received by the controller after the acceptable output has been achieved. In SoFT modules that act as clients, the acceptable outputs are requests to external servers. These servers may be SoFT modules or single-version components. The controller keeps a list of past acceptable outputs along with their respective replies and compensation messages. This list is used to send the external reply and the respective compensation message to versions whose outputs have arrived after the acceptable output has been agreed upon by the assessment components but before the deadline used by the controller has expired. The comparison between new messages and past acceptable outputs is done by the matching voter.

The controller of SoFT modules with versions that communicate via multiple entry/exit points is made of controllers identical to the one depicted in figure 10.1 that execute concurrently. Each of these controllers is linked to a set of equivalent version entry/exit points. They may have different configurations in order to cater for diverse assessment criteria.

Hardware fault-tolerance is automatically provided to versions and compensation modules that are instantiated at different nodes of the network. The remaining components of the SoFT model are made tolerant to hardware faults by means of well-known fault-tolerance techniques based on identical replicas (section 2.3).

## 10.2 Configurations of the SoFT Model

Software modules can play five different roles, as far as their inputs and outputs are concerned. They are: producer, consumer, client, server, and filter. The versions can execute either sequentially or concurrently. The former is adopted only in server
modules in order to save computational resources. Otherwise the latter should be used to improve response time. The outputs of the versions may be sent to the assessment components either in a static order or in a dynamic (first-come-first-served) fashion. Such orders are determined by issues such as the quality of the output and the cost of producing it.

SoFT controllers must be capable of dealing with all possible combinations of role, execution mode, assessment order, and assessment strategy. The number of possible combinations is thirty-five (chapter 7). Figure 10.2, which is a reproduction of figure 7.2, disregards the existence or absence of input validation, thus reducing the number of possible combinations to the twenty-two depicted. Practical experience has shown that it is feasible to design and implement a single, generic controller which is capable of dealing with all thirty-five possible combinations. However, the fraction of this code which would be unused at a particular configuration would range from 32% to 63%. Therefore, it is preferable to develop a set of different controllers.
Five semi-specialised controllers suffice to realise all possible combinations of role and execution mode. The fraction of unused lines of code shared by these controllers would then fall to the range of 7% to 19%. The necessary and sufficient number of more specialised controllers which leave only the assessment strategy as a variable is eight. The percentage of unused lines of code would fall even further: to 6% in the worst case and to none in the best cases.

10.3 Final Remarks

The framework for structuring diversely designed software developed in this research aims to reduce the time and cost of producing software fault-tolerant applications. In order to achieve this goal, the SoFT framework is based on reusable components. According to this approach, a fault-tolerant software module should be functionally replaceable by any of its diversely designed versions, without requiring either the modules that communicate with it, or the versions, to be modified. It has been shown in this research that history-independent modules fulfil this requirement, but history-dependent modules do not. History-dependent versions usually need to recover from an error before they resume their normal computation. The SoFT model provides the means by which this can be attained, but transparency must be sacrificed. Hence, history-dependent modules usually need to be modified in order to be configured as versions within the SoFT framework.

The size of the SoFT controller implemented in this work has been found to be significantly greater than the size of implementations of other components of the framework, particularly assessment algorithms. As a result, the importance of configuring the control functions in a separate component has been demonstrated. This is a fundamental principle that must be followed by any software fault-tolerance model. In addition, it has been established that only a small number of different controllers are ever needed.
Chapter 10. Conclusions

It has been shown in this research that, qualitatively, the reusability of assessment components, error handling modules, and control functions is improved by implementing them as separate modules, thus reducing the cost of new applications and increasing their reliability. The experiment in air traffic control described in chapter 8 has provided clear indication of the applicability of the SoFT framework for structuring fault-tolerant software modules. In order to measure the benefits of the SoFT model quantitatively and comprehensively, it would be necessary to carry out a large scale experiment, as proposed in section 10.4. Such an experiment has been left out of the scope of this work due to its cost, particularly in terms of human resources.

10.4 Future Research

A significant piece of research that would extend the work presented in this thesis is an experiment in which different models for building software fault-tolerant systems, including SoFT, would be compared. In brief, such an experiment should consist of developing the same software system in each of the approaches to be assessed. It would require some metrics for evaluating the cost of each system as well as a means by which the reliability of their modules could be compared.

Another topic in which further research would broaden the results of this work is on-line re-configuration. Versions should not be removed from the system when an error in them is detected, because they may be successful under inputs in which others fail. However, faults should be fixed off-line and the version replaced by the new release. Availability requirements may demand that the fault-tolerant module which contains the faulty version be re-configured without shutting it down. Therefore, it is appropriate to investigate whether existing mechanisms and policies for on-line replacement of software modules [GJ93] can be used or if new rules have to be specially created for the SoFT framework.
Chapter 10. Conclusions

Object-oriented systems is an important area in Computer Science nowadays. Object-oriented languages support the features that are required of a programming paradigm to realise the SoFT model, e.g., data encapsulation, and communication by message passing. In addition, they cater for reusable software components. Therefore, the possibility of writing SoFT modules in object-oriented languages should be examined. It is specially valuable to investigate how other features of object-oriented systems, such as inheritance, could be beneficial to SoFT. SoFT versions should not inherit any of their properties from the same class, because it could result in common failure modes. However, inheritance may be a powerful tool to configure fault-tolerant modules based on the SoFT framework.

It has been proposed recently that formal verification should become standard for safety critical applications [LR93]. Formal specification and validation methods have limited application in large systems. However, they can be combined with the SoFT framework in order to ensure the correctness of the components used to provide fault tolerance to the diversely designed versions. Hence, a study of the adequacy of the SoFT framework for the application of formal specification and validation methods is of relevance to the development of highly reliable software systems.
Appendix A

Specification of the SoFT Model

This appendix contains the specification of a SoFT module expressed in a pseudo-language. The external module (section A.3.1) that communicates with the SoFT module acts as a client and, therefore, the versions are to play the role of servers. An example of one of the versions is represented in section A.3.2. The algorithms of the remaining versions ought to be dissimilar, but their interfaces must be identical to the one shown. There is an example of each of the components of the SoFT framework. One should notice, however, that with the exception of the components specified in section A.2, the remaining ones may (some must) vary from application to application. The main purpose of representing them in this appendix is to illustrate their interfaces and to give a general overview of the SoFT model in an abstract notation.

Section A.1 specifies a SoFT module which plays the role of a server. The SoFT controller is configured to forward the external request to the versions concurrently, and to read their replies and to send them to the output assessment components in a first-come-first-served order. The version outputs are specified to be assessed by the voter first and then, if a reliable output is achieved, such an output is submitted to the conformity test. However, none of the components specified in this appendix would need changing if the execution mode of the versions were sequential, or if their outputs were read in a static order, or if their outputs were submitted to the
conformity test before been sent to the voter.

Since the versions (section A.3.2) communicate via a single port, no token administrator needs to be configured in the SoFT module specified in section A.1. However, the token administrator is specified in section A.2.1 in order for all SoFT components to be covered in this appendix.

Section A.5 summarises the types of messages exchanged between the controller and the other components of the SoFT model.

The notation used in the modules specified in this appendix and in figures throughout the thesis is described in section A.6 both syntactically and semantically.

A.1 Definition of the Model

The client-server configuration used as an example in this appendix is depicted in figure A.1. Modules command and alarm have not been specified in this appendix. Server $\Sigma$ is a composite component based on the SoFT framework. The internal structure of $\Sigma$ is shown in figure A.2.
Appendix A. Specification of the SoFT Model

Figure A.1: A Server SoFT Module in a Client-Server Configuration

Figure A.2: SoFT Server $\Sigma$
Appendix A. Specification of the SoFT Model

```plaintext
type Role_Type = { PRODUCER or CONSUMER or CLIENT or SERVER };
type Execution_Mode = { SEQUENTIAL or CONCURRENT };  
type Assessment_Mode = { STATIC or DYNAMIC };  
type Assessment_Order = { VOTER_FIRST or CT_FIRST };  

simple-component configure( )

  initiate
  create supervisor( ) at node_A;
  create Σ( node node_controller := node_B,
              node node_version_1 := node_X,
              node node_version_2 := node_Y,
              node node_version_3 := node_Z );
  link supervisor.target to (command module).(port);
  link supervisor.alarm to (alarm module).(port);
  link supervisor.supervision to controller.σ;
end

composite-component Σ( node node_controller,
                         node node_version_1,
                         node node_version_2,
                         node node_version_3 )

reply-port σ;

initiate

  create controller( Role_Type role := SERVER,
                    integer n_of_versions := 3,
                    integer n_of_ehms := 1,
                    Execution_Mode execution_mode := CONCURRENT,
                    Assessment_Mode assessment_mode := DYNAMIC,
                    Assessment_Order assessment_order := VOTER_FIRST,
                    time ext_timeout := 30 seconds ,
                    time iv_timeout := 30 seconds ,
                    time ct_timeout := 30 seconds ,
                    time v_timeout := 30 seconds ,
                    time mv_timeout := 30 seconds ,
                    time version_timeout := 30 seconds ,
                    time ehm_timeout := 30 seconds ) at node_controller;
create ivm( ) at node_controller;
create ct( ) at node_controller;
create v( ) at node_controller;
create mv( ) at node_controller;
create ehm( ) at node_controller;
create sensor_monitor( ) at node_version_1;
create cm( ) at node_version_1;
create (second version) at node_version_2;
create (second compensation module) at node_version_2;
```

```
create (third version) at node_version.3;
create (third compensation module) at node_version.3;

link σ to controller.ξ;
link controller.aiv to ivm.check;
link controller.aet to ct.service;
link controller.αv to voter.service;
link controller.αμυ to (matching voter).(port);
link controller.ε[0] to ehm.service;
link controller.π[1] to sensor.monitor.sensor.control;
link controller.π[2] to (version 2).(port);
link controller.π[3] to (version 3).(port);
link controller.ρ[1] to cm.feedback;
link controller.ρ[2] to (cm 2).(port);
link controller.ρ[3] to (cm 3).(port);
end

A.2 The Controller

type Role_Type = { PRODUCER or CONSUMER or CLIENT or SERVER };
type Execution_Mode = { SEQUENTIAL or CONCURRENT };
type Assessment_Mode = { STATIC or DYNAMIC };
type Assessment_Order = { VOTER FIRST or CT FIRST };
type Msg_Type = bag of universal;
type Status_Type = { SUCCESS or FAILURE or INDECISION };
type Communication_Status = { SUCCESS or FAILURE or ⋅ ⋅ ⋅ };
type Past_Exports = { Msg_Type out_msg and Msg_Type in_msg and
bag of integer index };
type Port = { input-port or output-port or request-port or reply-port };

simple-component controller( Role_Type role,
    integer n_of_versions,
    integer n_of_ehms,
    Execution_Mode execution_mode,
    Assessment_Mode assessment_mode,
    Assessment_Order assessment_order,
    time ext_timeout,
    time iv_timeout,
    time ct_timeout,
    time v_timeout,
    time mv_timeout,
    time version_timeout,
    time ehm_timeout )
if role = PRODUCER →
  output-port ξ;
  bag of input-port π[n.of.versions];
  bag of input-port ρ[n.of.versions];
else if role = CONSUMER →
  input-port ξ;
  bag of output-port π[n.of.versions];
  bag of output-port ρ[n.of.versions];
else if role = CLIENT →
  request-port ξ;
  bag of reply-port π[n.of.versions];
  bag of reply-port ρ[n.of.versions];
else if role = SERVER →
  reply-port ξ;
  bag of request-port π[n.of.versions];
  bag of request-port ρ[n.of.versions];
fi
request-port α_{uv};
request-port α_{ct};
request-port α_{su};
request-port α_{mu};
bag of request-port ε[n.of.ehms];
request-port τ_{get};
output-port τ_{free};

state
  bag of Past_Exports past_exports;

initiate
  past_exports := θ;

do true →
  if role = PRODUCER →
    producer( );
  elseif role = CONSUMER →
    consumer( );
  elseif role = CLIENT →
    client( );
  elseif role = SERVER →
    server( );
  fi
od
end
Appendix A. Specification of the SoF T Model

function producer( )
    boolean voter_present := initiate_voter( );
    Status_Type out_assess := conc_out_protocol( voter_present,
        Status_Type status_of,
        Msg_Type successful_msg,
        integer ehm_id,
        Msg_Type error_msg );
    if out_assess = SUCCESS ->
        send \( \xi \).plain ( Msg_Type successful_msg );
        compensation_protocol( status_of, successful_msg, Msg_Type reply_msg := 0 );
    elseif out_assess = FAILURE ->
        ehm.protocol( ehm_id, error_msg );
    fi
end

function consumer( )
    receive at \( \xi \)
        case communication_error( Communication_Status code ) ->
            ehm.protocol( integer ehm_id := 0,
                Msg_Type error_msg := "Communication error"+code+" at \( \xi \) " );
        case communication_ok( Msg_Type external_msg ) ->
            Status_Type iv_status := iv_protocol( external_msg, ehm_id, error_msg );
            if iv_status = FAILURE ->
                ehm.protocol( ehm_id, error_msg );
            elseif iv_status = SUCCESS ->
                get_token( );
                for integer i := 1 to n_of_versions ->
                    send \( \pi[i] \).plain ( external_msg );
                af
                free_token( );
            fi
        within eternity ;
    end
function client( )

boolean voter_present := initiate_voter( );
Status_Type out_assess := conc_out_protocol( voter_present,
  Status_Type status_of,
  Msg_Type successful_msg,
  integer ehm_id,
  Msg_Type error_msg );

if out_assess = SUCCESS →
  request ξplain( Msg_Type successful_msg )
  case communication_error( Communication_Status code ) →
  ehm_protocol( integer ehm_id := 0,
    Msg_Type error_msg := "Communication error"+code+" at ξ" );
  case communication_ok( Msg_Type reply_msg ) →
  Status_Type iv_status := iv_protocol( reply_msg, ehm_id, error_msg );
  if iv_status = FAILURE →
    ehm_protocol( ehm_id, error_msg );
  elseif iv_status = SUCCESS →
    get_token( );
    fa integer i := 1 to n_of_versions →
      reply π[i].plain( reply_msg );
    af
    free_token( );
  f
  within ext_timeout;
  compensation_protocol( status_of, successful_msg, reply_msg );
  elseif out_assess = FAILURE →
    ehm_protocol( ehm_id, error_msg );
  fi
end
function server()
    receive at ξ
        case communication_error( Communication_Status code ) →
            ehm_protocol( integer ehm_id := 0,
                Msg_Type error_msg := "Communication error"+code+" at ξ" );
        case communication_ok( Msg_Type external_msg ) →
            Status_Type iv_status := iv_protocol( external_msg, ehm_id, error_msg );
                if iv_status = FAILURE →
                    ehm_protocol( ehm_id, error_msg );
            elseif iv_status = SUCCESS →
                if execution_node = SEQUENTIAL →
                    Status_Type request_assess := seq_req_protocol( Msg_Type external_msg,
                                Status_Type status_of,
                                Msg_Type successful_reply,
                                integer ehm_id,
                                Msg_Type error_msg );
                elseif execution_node = CONCURRENT →
                    Status_Type request_assess := conc_req_protocol( Msg_Type external_msg,
                                Status_Type status_of,
                                Msg_Type successful_reply,
                                integer ehm_id,
                                Msg_Type error_msg );
                fi
                if out_assess = SUCCESS →
                    reply ξ_plain( Msg_Type successful_reply );
                    compensation_protocol( status_of, successful_reply, Msg_Type ext_msg := 0 );
                elseif out_assess = FAILURE →
                    ehm_protocol( ehm_id, error_msg );
                fi
            fi
        fi
    within eternity;
end
function seq_req_protocol(Msg_Type external_request, Status_Type status_of, Msg_Type successful_reply, integer ehm_id, Msg_Type error_msg ); return Status_Type

boolean voter_present := initiate_voter( );
if voter_present = true →
  if assessment_order = VOTER_FIRST →
    seq_req_protocol := sequential_voter_first( external_request, status_of, successful_reply, ehm_id, error_msg );
  elseif assessment_order = CT_FIRST →
    seq_req_protocol := sequential_ct_first( external_request, status_of, successful_reply, ehm_id, error_msg );
  fi
elseif voter_present = false →
  seq_req_protocol := sequential_ct_only( external_request, status_of, successful_reply, ehm_id, error_msg );
fi
end

function conc_req_protocol(Msg_Type external_request, Status_Type status_of, Msg_Type successful_reply, integer ehm_id, Msg_Type error_msg ); return Status_Type

get_token( );
fa integer i := 1 to n_of_versions →
  request x[i].plain( external_request )
  within 0;
af
free_token( );
boolean voter_present := initiate_voter( );
conc_req_protocol := conc_out_protocol( voter_present, status_of, successful_reply, ehm_id, error_msg );
end
function conc.out.protocol( boolean voter.psnent,
    Status.Type status.of,
    Msg.Type successful.msg,
    integer ehm_id,
    Msg.Type error.msg ): return Status.Type

    if voter.present = true →
        if assessment.mode = STATIC →
            if assessment.order = VOTER.FIRST →
                conc.out.protocol := static.voter.first( status.of, successful.msg,
                    ehm_id, error.msg );
            elseif assessment.order = CT.FIRST →
                conc.out.protocol := static.ct.first( status.of, successful.msg,
                    ehm_id, error.msg );
        fi
    elseif assessment.mode = DYNAMIC →
        if assessment.order = VOTER.FIRST →
            conc.out.protocol := dynamic.voter.first( status.of, successful.msg,
                ehm_id, error.msg );
        elseif assessment.order = CT.FIRST →
            conc.out.protocol := dynamic.ct.first( status.of, successful.msg,
                ehm_id, error.msg );
        fi
    fi
elseif voter.present = false →
    if assessment.mode = STATIC →
        conc.out.protocol := static.ct.only( status.of, successful.msg,
            ehm_id, error.msg );
    elseif assessment.mode = DYNAMIC →
        conc.out.protocol := dynamic.ct.only( status.of, successful.msg,
            ehm_id, error.msg );
    fi
fi
end
Appendix A. Specification of the SoFT Model

function static_voter_first( Status.Type status_of,  
    Msg.Type successful_msg,  
    integer ehm_id,  
    Msg.Type error_msg ): return Status.Type  

time time_out := eternity ;  
integer index := 1;  
Status.Type decision := INDECISION;  
do index ≤ n_of_versions and decision = INDECISION →  
receive at x[index]  
    case late_msg( Msg.Type version_msg ) →  
        matching_voter_protocol( index, version_msg );  
    case communication_error( Communication.Status code ) →  
        decision := voter_protocol_failure( index, ehm_id, ehm_msg );  
        index := index + 1;  
        if time_out = eternity ;  
            time_out := version_timeout;  
        fi  
    with in timeout;  
    od  
if decision = SUCCESS →  
    static_voter_first := ct_protocol( successful_msg, ehm_id, ehm_msg );  
elsif decision = INDECISION or decision = FAILURE →  
    static_voter_first := FAILURE;  
fi  
end
function static.ct.first( Status.Type status.of,
    Msg.Type successful.msg,
    integer ehm_id,
    Msg.Type error.msg ): return Status.Type 

    time time.out := eternity ;
    integer index := 1;
    Status.Type decision := INDECISION;
    do index ≤ n.of.versions and decision = INDECISION ->
        receive at tt[index]
            case late.msg( Msg.Type version.msg ) ->
                matching_voter_protocol( index, version.msg );
            case communication_error( Communication_Status code ) ->
                decision := voter_protocol_failure( index, ehm_id, ehm_msg );
                index := index + 1;
                if time.out = eternity ;
                    time.out := version.timeout;
                    fi
                end
            case version.msg( Msg.Type version.msg ) ->
                decision := ct_protocol( version.msg, ehm_id, ehm_msg );
                if decision = SUCCESS ->
                    decision := voter_protocol_success( index, version.msg,
                        status.of, successful.msg,
                        ehm_id, ehm_msg );
                    end
            elseif decision = FAILURE ->
                decision := voter_protocol_failure( index, ehm_id, ehm_msg );
                fi
                index := index + 1;
                if time.out = eternity ;
                    time.out := version.timeout;
                    fi
                end
            within timeout;
        od
    if decision = SUCCESS ->
        static.ct.first := SUCCESS;
    elseif decision = INDECISION or decision = FAILURE ->
        static.ct.first := FAILURE;
    fi
end
Appendix A. Specification of the SoFT Model

function dynamic_voter_first(Status_Type status_of,
    Msg_Type successful_msg,
    integer ehm_id,
    Msg_Type error_msg): return Status_Type

bag of port in := \pi[1] + \cdots + \pi[n_of_versions];
time time_out := eternity;
Status_Type decision := INDECISION;
do in \neq \emptyset and decision = INDECISION ->
    receive at any of in
        case late_msg(Msg_Type version_msg) ->
            matching_voter_protocol(index, version_msg);
        case communication_error(Communication_Status code) ->
            decision := voter_protocol_failure(index_of(sender), ehm_id, ehm_msg);
            in := in \setminus sender;
            if time_out = eternity;
                time_out := version_timeout;
            fi
        fi
        case version_msg(Msg_Type version_msg) ->
            decision := voter_protocol_success(index_of(sender), version_msg,
                status_of, successful_msg,
                ehm_id, ehm_msg);
            in := in \setminus sender;
            if time_out = eternity;
                time_out := version_timeout;
            fi
        within timeout;
    od
    if decision = SUCCESS ->
        dynamic_voter_first := ct_protocol(successful_msg, ehm_id, ehm_msg);
    elseif decision = INDECISION or decision = FAILURE ->
        dynamic_voter_first := FAILURE;
    fi
end
Appendix A. Specification of the SoFT Model

function dynamic_ct_first( Status_Type status_of,
                         Msg_Type successful_msg,
                         integer ehm_id,
                         Msg_Type error_msg ): return Status_Type

bag of port in := \pi[1] + \cdots + \pi[n_of_versions];
time time_out := eternity ;
Status_Type decision := INDECISION;
do in \neq \emptyset and decision = INDECISION –>
receive at any of in
  case late_msg( Msg_Type version_msg ) ->
    matching_voter_protocol( index, version_msg );
  case communication_error( Communication_Status code ) ->
    decision := voter_protocol_failure( index_of( sender ), ehm_id, ehm_msg );
in := in \setminus \text{sender} ;
  if time_out = eternity ;
    time_out := version_timeout;
  fi
  case version_msg( Msg_Type version_msg ) ->
    decision := ct_protocol( version_msg, ehm_id, ehm_msg );
  if decision = SUCCESS ->
    decision := voter_protocol_success( index_of( sender ), version_msg,
                                        status_of, successful_msg,
                                        ehm_id, ehm_msg );
  elseif decision = FAILURE ->
    decision := voter_protocol_failure( index_of( sender ), ehm_id, ehm_msg );
  fi
  in := in \setminus \text{sender} ;
  if time_out = eternity ;
    time_out := version_timeout;
  fi
  within timeout;
od
if decision = SUCCESS ->
dynamic_ct_first := SUCCESS;
elseif decision = INDECISION or decision = FAILURE ->
dynamic_ct_first := FAILURE;
fi
end
function static_ct_only( Status_Type status_of,
    Msg_Type successful_msg,
    integer ehm_id,
    Msg_Type error_msg ); return Status_Type

    for integer i := 1 to n.of_versions →
        status_of[index] := INDECISION;
    af
    time time_out := eternity ;
    integer index := 1;
    Status_Type decision := FAILURE;
    do index ≤ n.of_versions and decision = FAILURE →
        receive at π[index]
            case late_msg( Msg_Type version_msg ) →
                matching_voter_protocol( index, version_msg );
            case communication_error( Communication_Status code ) →
                status_of[index] := FAILURE;
                index := index + 1;
                if time_out = eternity ;
                    time_out := version_timeout;
                fi
            case version_msg( Msg_Type version_msg ) →
                decision := ct_protocol( version_msg, ehm_id, ehm_msg );
                status_of[index] := decision;
                index := index + 1;
                if time_out = eternity ;
                    time_out := version_timeout;
                fi
        within timeout;
    od
    if decision = SUCCESS →
        static_ct_only := SUCCESS;
        successful_msg := version_msg;
    elseif decision = FAILURE →
        static_ct_only := FAILURE;
    fi
end
Appendix A. Specification of the SoFT Model

function dynamic.ct.only( Status.Type status.of, 
   Msg.Type successful.msg, 
   integer ehm.id, 
   Msg.Type error.msg ): return Status.Type

fa integer i := 1 to n.of.versions →
   status.of[index] := INDECISION;
af
bag of port in := \pi[1] + \cdots + \pi[n.of.versions];
time time.out := eternity ;
Status.Type decision := FAILURE;
do in \neq \emptyset and decision = FAILURE →
   receive at any of in
      case late/msg( Msg.Type version.msg ) →
         matching.voter.protocol( index, version.msg );
      case communication.error( Communication.Status code ) →
         status.of[index.of( sender )] := FAILURE;
in := in \ominus sender ;
      if time.out = eternity ;
         time.out := version.timeout;
      fl
      case version/msg( Msg.Type version.msg ) →
         decision := ct.protocol( version.msg, ehm.id, ehm.msg );
         status.of[index.of( sender )] := decision;
in := in \ominus sender ;
      if time.out = eternity ;
         time.out := version.timeout;
      fl
   within timeout;
od
if decision = SUCCESS →
   dynamic.ct.only := SUCCESS;
   successful.msg := version.msg;
elseif decision = FAILURE →
   dynamic.ct.only := FAILURE;
fi
end
function sequential_voter_first( Msg.Type external_request,
    Status.Type status_of,
    Msg.Type successful_reply,
    integer ehm_id,
    Msg.Type error_msg )
    return Status.Type

time time_out := eternity ;
integer index := 1;
Status.Type decision := INDECISION;
get_token( );
do index ≤ n.of_versions and decision = INDECISION →
    request at \( \pi[\text{index}].plain \) ( external_request )
    case late_msg( Msg.Type version_msg ) →
        matching_voter_protocol( index, version_msg );
    case communication_error( Communication.Status code ) →
        decision := voter_protocol_failure( index, ehm_id, ehm_msg );
        index := index + 1;
        if time_out = eternity ;
            time_out := version_timeout;
        fi
        decision := INDECISION or decision = FAILURE →
    fi
    within timeout;
od
free_token( );
if decision = SUCCESS →
    sequential_voter_first := ct_protocol( successful_reply, ehm_id, ehm_msg );
elseif decision = INDECISION or decision = FAILURE →
    sequential_voter_first := FAILURE;
fi
end
Appendix A. Specification of the SoFT Model

function sequential.ct.first(Msg.Type external.request, Status.Type status.of, Msg.Type successful.reply, integer ehm.id, Msg.Type error.msg): return Status.Type
time time.out := eternity;
integer index := 1;
Status.Type decision := INDECISION;
get.token();
do index \leq n.of.versions and decision = INDECISION →
request at π[index].plain( external.request )
case late.msg( Msg.Type version.msg ) →
  matching.voter.protocol( index, version.msg );
case communication.error( Communication.Status code ) →
  decision := voter.protocol.failure( index, ehm.id, ehm.msg );
  index := index + 1;
  if time.out = eternity;
    time.out := version.timeout;
  fi
  case version.msg( Msg.Type version.msg ) →
    decision := ct.protocol( version.msg, ehm.id, ehm.msg );
    if decision = SUCCESS →
      decision := voter.protocol.success( index, version.msg, status.of, successful.reply, ehm.id, ehm.msg );
    elseif decision = FAILURE →
      decision := voter.protocol.failure( index, ehm.id, ehm.msg );
    fi
    index := index + 1;
    if time.out = eternity;
      time.out := version.timeout;
    fi
  within timeout;
od
free.token();
if decision = SUCCESS →
  sequential.ct.first := SUCCESS;
elseif decision = INDECISION or decision = FAILURE →
  sequential.ct.first := FAILURE;
fi
end
Appendix A. Specification of the SoFT Model

function sequential_ct_only( Msg.Type external_request,
    Status.Type status_of,
    Msg.Type successful_reply,
    integer ehm_id,
    Msg.Type error_msg ): return Status.Type

    fa integer i := 1 to n_of_versions →
        status_of[index] := INDECISION;
    af
    time timeout := eternity ;
    integer index := 1;
    Status.Type decision := FAILURE;
    get_token( );
    do index ≤ n_of_versions and decision = FAILURE →
        request at π[index].plain ( external_request )
        case late_msg( Msg.Type version_msg ) →
            matching_voter_protocol( index, version_msg );
        case communication_error( Communication.Status code ) →
            status_of[index] := FAILURE;
            index := index + 1;
            if timeout = eternity ;
                timeout := version_timeout;
            fl
        case version_msg( Msg.Type version_msg ) →
            decision := ct.protocol( version_msg, ehm_id, ehm_msg );
            status_of[index] := decision;
            index := index + 1;
            if timeout = eternity ;
                timeout := version_timeout;
            fl
        within timeout;
    od
    free_token( );
    if decision = SUCCESS →
        sequential_ct_only := SUCCESS;
        successful_msg := version_msg;
    elseif decision = FAILURE →
        sequential_ct_only := FAILURE;
    fl
end
function index_of( Port port_id ): return integer
if (port_id ∈ a bag of Port) →
  index_of := (index of port_id);
else →
  index_of := (invalid index);
fi
end

function matching_voter_protocol( integer version_id, Msg.Type version_msg )
request at α_{mv}.init_msg( integer n_versions := 2 )
within 0;
Msg.Type late_out_msg := first_out_match( version_id );
Msg.Type late_in_msg := first_in_match( version_id );
request at α_{mv}.version_msg( integer dummy_version_id_1 := 1, late_out_msg )
within mv.timeout;
request at α_{mv}.version_msg( integer dummy_version_id_2 := 2, version_msg )
case success( Status.Type dummy_status_of,
  Msg.Type dummy_successful_msg ) →
  if role = PRODUCER or role = SERVER →
    send ρ[version_id].success( );
  elseif role = CLIENT →
    reply π[version_id].plain( late_in_msg );
    send ρ[version_id].success( );
  fi
  case failure( integer dummy_error_id, Msg.Type dummy_error_msg ) →
    send ρ[version.id].error( late_out_msg );
  within mv.timeout;
end

function first_out_match( integer version_id ): return Msg.Type
first_out_match := (out_msg of past.exports associated with
the first occurrence of version_id);
end

function first_in_match( integer version_id ): return Msg.Type
first_in_match := (in_msg of past.exports associated with
the first occurrence of version_id);
end
function initiate_voter( ): return boolean
    request at αe.init_msg( n.of.versions );
    case voter_exists( ) →
        initiate_voter := true ;
    case voter_does_not_exist( ) →
        initiate_voter := false ;
    within 0;
end

function voter_protocol_success( integer version_id, 
    Msg.Type version_msg, 
    Status.Type status_of, 
    Msg.Type successful_msg, 
    integer ehm_id, 
    Msg.Type ehm_msg ): return Status.Type
    request at αe.version_msg( version_id, version_msg )
    case success( status_of, successful_msg ) →
        voter_protocol_success := SUCCESS;
    case indecision( ) →
        voter_protocol_success := INDECISION;
    case failure( ehm_id, error_msg ) →
        voter_protocol_success := FAILURE;
    within voter_timeout;
end

function voter_protocol_failure( integer version_id, 
    integer ehm_id, 
    Msg.Type ehm_msg ): return Status.Type
    request at αe.fail_msg( version_id )
    case indecision( ) →
        voter_protocol_failure := INDECISION;
    case failure( ehm_id, error_msg ) →
        voter_protocol_failure := FAILURE;
    within voter_timeout;
end

function ct_protocol( Msg.Type version_msg 
    integer ehm_id, Msg.Type ehm_msg ): return Status.Type
    request at αct.a_msg( version_msg )
    case inexist( ) →
        ct_protocol := SUCCESS;
    case success( ) →
        ct_protocol := SUCCESS;
    case failure( ehm_id, ehm_msg ) →
        ct_protocol := FAILURE;
    within ct_timeout;
end
Appendix A. Specification of the SoFT Model

function compensation_protocol( Status_Type status_of,
                             Msg_Type out_msg, Msg_Type in_msg )

    bag of integer late_versions := 0;
    fa integer index := 1 to n.of.versions →
        if status_of[index] = SUCCESS →
            send ρ[index].success( );
        elseif status_of[index] = FAILURE →
            send ρ[index].failure( out_msg );
        elseif status_of[index] = INDECISION →
            late_versions := late_versions + index;
        fi
    fi
    if late_versions ≠ 0 →
        past_exports := past_exports + {out_msg, in_msg, late_versions};
    fi
end

function ehm_protocol( integer ehm_id, Msg_Type error_msg )

    request at e[ehm_id].error_handling( error_msg )
        case inexistent( ) →
            skip
        case external( Msg_Type ehm_reply_msg ) →
            reply ξ.plain ( ehm_reply_msg );
        case compensation( Msg_Type ehm_reply_msg ) →
            fa integer i := 1 to n.of.versions →
                send ρ[i].general_failure( ehm_reply_msg );
            af
        case both( Msg_Type ehm_reply_msg ) →
            reply ξ.plain ( ehm_reply_msg );
            fa integer i := 1 to n.of.versions →
                send ρ[i].general_failure( ehm_reply_msg );
            af
        case none( ) →
            skip
        case another.ehm( integer ehm_id_2, Msg_Type ehm_reply_msg ) →
            ehm.protocol( ehm_id_2, ehm_reply_msg );
    within ehm.timeout;
end
function iv_protocol(Msg_Type external_msg, integer ehm_id, Msg_Type ehm_msg): return Status_Type
request at α_{iv_a_msg}(external_msg)
case inexistent():
  iv_protocol := SUCCESS;
case success():
  iv_protocol := SUCCESS;
case failure(ehm_id, ehm_msg):
  iv_protocol := FAILURE;
within iv_timeout;
end

function get_token():
request at τ_{get.get_token()}
case token_administrator.inexistent():
  skip;
case token_granted():
  skip;
within eternity;
end

function free_token():
send at τ_{free.free_token()};
end

A.2.1 The Token Administrator

simple-component token_administrator():
  reply-port token_request;
  input-port token_release;
  do true →
    receive at token_request
      case get_token():
        skip;
      within eternity;
      reply token_request.token_granted();
    receive at token_release
      case free_token():
        skip;
      within eternity;
  od
end
Appendix A. Specification of the SoFT Model

A.3 External Modules and Versions

A.3.1 An External Module

simple-component supervisor()

reply-port target;
request-port supervision;
output-port alarm;

state
real currentsetting;

do true
receive at target
  case new_target(real optimum_value) →
    currentsetting := calculate_setting(optimum_value);
    request at supervision.new_setting(currentsetting)
      case match( ) →
        reply target.success( );
      case error(real error) →
        reply target.error(error);
      case timeout( ) →
        reply target.timeout.error( );
        send alarm.non-responding.sensor( );
      within 15 seconds ;
    case timeout( ) →
      request at supervision.sensor_reading( )
      case value(real sensor_reading) →
        if sensor_reading ≠ currentsetting →
          send alarm.reading.offset_by(sensor_reading − currentsetting);
        fi
      case timeout( ) →
        send alarm.non-responding.sensor( );
      within 3 minutes ;
    od
end

function calculate_setting(real optimum_value): return real
  calculate_setting := (offset to be applied on the physical device);
end
A.3.2 A Version

simple-component sensor_monitor()

reply-port sensor_control;

do true →
  receive at sensor_control
  case new_setting( real value ) →
    set_actuator( value );
    real sensor_reading := read_sensor( );
    if sensor_reading = value →
      reply sensor_control.match( );
    else →
      reply sensor_control.error( sensor_reading − value );
    fi
  case sensor_reading( ) →
    real sensor_reading := read_sensor( );
    reply sensor_control.value( sensor_reading );
  within eternity ;
  od
end

function set_actuator( real setting )
  (physically acts on the mechanical device according to “setting”);
end

function read_sensor(): return real
  read_sensor := (physical sensor reading);
end
Appendix A. Specification of the SoFT Model

A.4 Other Components

A.4.1 An Input Validation Module

simple-component ivm()

reply-port check;

do true →
receive at check
  case a_msg( real sensor_setting ) →
    if valid_setting( sensor_setting ) = true →
      reply check.success( );
    else →
      reply check.error( );
  fi
within eternity ;
od
end

function valid_setting( real sensor_setting ): return boolean
if (minimum setting < sensor_setting < maximum setting ) →
  valid_setting := true ;
else →
  valid_setting := false ;
fi
end
A.4.2 A Conformity Test

```plaintext
simple-component ct()

  reply-port service;

  do true ->
    receive at service
      case a_msg( real sensor_reading ) ->
        if within_range( sensor_reading ) = true ->
          reply service.success( );
        else ->
          reply service.error( );
        fi
        within eternity;
      od
  od
end

function within_range( real sensor_reading ): return boolean
  if (minimum reading ≤ sensor_reading ≤ maximum reading ) ->
    within_range := true ;
  else ->
    within_range := false ;
  fi
end
```
Appendix A. Specification of the SoFT Model

A.4.3 A Majority Voter

type Msg_Type = bag of universal;
type Status_Type = { SUCCESS or FAILURE or INDECISION };

simple-component voter()

   reply-port service;

   state

      integer n_of_versions;
      bag of Msg_Type version_msg;
      bag of Status_Type status_of;

   do true ➔

      receive at service

      case init_msg(integer n_versions) ➔
         n_of_versions := n_versions;
         version_msg := empty;
         for integer i := 1 to n_of_versions ➔
            status_of[i] := INDECISION;
         af

      case fail_msg(integer version) ➔
         status_of[version] := FAILURE;
         if majority_possible() = true ➔
            reply service.indecision();
         else ➔
            reply service.error(integer ehm_id := 0, Msg_Type ehm_msg :=
                "Communication error with most versions");
         fl

      case version_msg(integer version, Msg_Type msg) ➔
         version_msg[version] := msg;
         status_of[version] := SUCCESS;
         if majority_found() = true ➔
            comp_and_mark( the_majority() );
            reply service.success(status_of, Msg_Type the_majority() );
         elseif majority_possible() = true ➔
            reply service.indecision();
         else ➔
            reply service.error(integer ehm_id := 0, Msg_Type ehm_msg :=
                "No agreement has been reached.");
         fi

      within eternity;
   od
end
function majority_possible( ): return boolean
    if (∃ a ∈ version_msg | count( a, version_msg ) +
        count( INDECISION, status_of ) ≥ \[\frac{n\text{-of\text{-}versions}+1}{2}\]) →
        majority_possible := true ;
    else
        majority_possible := false ;
    fi
end

function the_majority( ): return Msg_Type
    if (∃ a ∈ version_msg | count( a, version_msg ) ≥ \[\frac{n\text{-of\text{-}versions}+1}{2}\]) →
        the_majority := (a);
    else
        the_majority := null ;
    fi
end

function majority_found( ): return boolean
    if (∃ a ∈ version_msg | count( a, version_msg ) ≥ \[\frac{n\text{-of\text{-}versions}+1}{2}\]) →
        majority_found := true ;
    else
        majority_found := false ;
    fi
end

function comp_and_mark( maj: Msg_Type )
    for integer i := 1 to n_of_versions →
        if status_of[i] = SUCCESS and version_msg[i] ≠ maj →
            status_of[i] := FAILURE;
        fi
    af
end

function count( elem: universal, set: bag of universal ): return integer
    count := 0;
    for integer i := 1 to sizeof( set ) →
        if set[i] = elem →
            count++;
        fi
    af
end
Appendix A. Specification of the SoFT Model

A.4.4 A Compensation Module

type Msg_Type = bag of universal;

simple-component cm()

input-port feedback;

do true →
   receive at feedback
   case success( ) →
      (no compensation procedure);
   case failure( Msg_Type successful_output ) →
      (perform compensation procedure based on successful_output);
   within eternity;
   od
end

A.4.5 An Error Handling Module

type Msg_Type = bag of universal;

simple-component ehm()

reply-port service;

do true →
   receive at service
   case error_handling( Msg_Type error_msg ) →
      reply service.external( error_msg );
   within eternity;
   od
end

A.5 Summary of Message Types

The types of the messages exchanged between the controller and the other components of the SoFT model and external modules are briefly described in this section.
The two types defined ahead will be used in the definition of the messages in this section.

\[
\begin{align*}
\text{type } \text{Msg.Type} &= \text{bag of universal}; \\
\text{type } \text{Status.Of} &= \{\text{SUCCESS or FAILURE or INDECISION}\};
\end{align*}
\]

Messages Exchanged with External Modules

All messages exchanged with external modules are treated by the controller as a collection of data of any type. Thus, in the abstract representation used in the thesis, messages exchanged with external modules can be defined as:

\[
\begin{align*}
\text{type } \text{to_external} &= \text{Msg.Type}; \text{acceptable message} \\
\text{type } \text{from_external} &= \text{Msg.Type}; \text{external module message}
\end{align*}
\]

Messages Exchanged with the Versions

Similarly, all messages exchanged with the versions are treated by the controller as a collection of data of any type. Thus, in the abstract representation used in the thesis, messages exchanged with external modules can be defined as:

\[
\begin{align*}
\text{type } \text{to_version} &= \text{Msg.Type}; \text{external module message} \\
\text{type } \text{from_version} &= \text{Msg.Type}; \text{version message}
\end{align*}
\]

Messages Exchanged with Compensation Modules

Messages exchanged with the compensation modules indicate either that the output issued by the respective version has been found to be acceptable or that it has disagreed from the acceptable one. Thus, the following are the two types of messages exchanged with the compensation modules:

\[
\begin{align*}
\text{type } \text{to_cm.success} &= \text{void} \\
\text{type } \text{to_cm.failure} &= \text{Msg.Type}; \text{acceptable message}
\end{align*}
\]
Appendix A. Specification of the SoFT Model

Messages Exchanged with the Input Validation Module

The controller sends the messages received from external modules to the input validation module, which replies either validating the message or rejecting it. Thus, the messages exchanged by the controller with the input validation module are:

\[\text{type to ivm} = \text{Msg.Type}; \text{external module message} \]
\[\text{type from ivm.success} = \text{void}; \]
\[\text{type from ivm.failure} = \text{void}; \]

Messages Exchanged with the Conformity Test

The conformity test acts on version outputs in the same way as the input validation module does on external messages. In fact, it is possible that input validation modules and conformity test are different instances of the same component. Thus, the messages exchanged by the controller with the conformity test are:

\[\text{type to ct} = \text{Msg.Type}; \text{version message or acceptable message} \]
\[\text{type from ct.success} = \text{void}; \]
\[\text{type from ct.failure} = \text{void}; \]

Messages Exchanged with the Voter

The controller informs the voter that a round of voting is to commence by sending to it an initiation message. The voter does not reply to the initiation message. Subsequently, it will send to the voter the outputs issued by the versions or a communication error indication. To each of these messages, the voter will reply either with a success message, or with an indecision message, or with a failure message. In brief:
Appendix A. Specification of the SoFT Model

\begin{verbatim}
type to_voter_initiate = integer; number of versions
type to_voter_version_msg = { integer version_id and Msg_Type version_msg }

type to_voter_faulty_version = integer version_id;
type from_voter_success = { bag of Status_Of status_list and
   Msg_Type acceptable_msg }

type from_voter_failure = { integer ehm_id and Msg_Type msg_to_ehm }

type from_voter_indecision = void;
\end{verbatim}

Messages Exchanged with the Matching Voter

The function of the matching voter is different from that of the voter. However, in
many cases, a voting module can be used as a voter, a matching voter or both. Thus,
the messages exchanged between the controller and the matching voter are analogous
to the ones between the controller and the voter. The difference is that the matching
voter collates on two messages only: the acceptable message and a late version
message. A communication error with a late version dispenses with communicating
with the matching voter. Hence, instead of six different types of messages, the
number of different types of messages exchanged between the controller and the
matching voter is five:

\begin{verbatim}
type to_mv_initiate = const 2;
type to_mv_version_msg = { integer altered_version_id and
   Msg_Type acceptable_or_version_msg }

type from_mv_success = { bag of Status_Of unused_status_list and
   Msg_Type unused_acceptable_msg }

type from_mv_failure = { integer unused_ehm_id and
   Msg_Type unused_msg_to_ehm }

type from_mv_indecision = void;
\end{verbatim}

Messages Exchanged with the Error Handling Modules

An error handling module is activated by the controller by sending to it a message
which the controller has received from the assessment components when they replied
with an error message. An error handling module reply, which contains a message
whose contents are not inspected by the controller, indicates one of five possible
addressees for such a message. Hence:

\[
\begin{align*}
\text{type } to\_ehm &= \text{Msg\_Type}; \text{\it message from assessment component} \\
\text{type } from\_ehm\_external &= \text{Msg\_Type}; \text{\it error message} \\
\text{type } from\_ehm\_internal &= \text{Msg\_Type}; \text{\it compensation message} \\
\text{type } from\_ehm\_both &= \text{Msg\_Type}; \text{\it error and compensation message} \\
\text{type } from\_ehm\_no\_component &= \text{void}; \\
\text{type } from\_ehm\_another\_ehm &= \{ \text{Msg\_Type message and integer ehm\_id} \}
\end{align*}
\]

### A.6 Notation

#### A.6.1 Semantics

This section contains the semantics of the specifications depicted in the earlier sections of this appendix and throughout the thesis. Structures used in the specifications that have a clear meaning on their own will be briefly presented. Only those which may lead to misunderstandings of their semantics will be more thoroughly explained.

**Type Definition**

The type definition in the specifications shown in this thesis are similar to the type definition in Pascal. Types can be defined by *enumeration* in the format:

\[
\begin{align*}
\text{definition : type } &\text{(type name)} = \{ \text{first item or } \cdots \text{ or last item} \} \\
\text{example : type } &\text{Execution\_Mode} = \{ \text{SEQUENTIAL or CONCURRENT} \};
\end{align*}
\]

Types representing *structures*, to use the C terminology, or *records*, to use the Pascal one, are defined in the format:

\[
\begin{align*}
\text{definition : type } &\text{(type name)} = \{ \text{first field and } \cdots \text{ and last field} \} \\
\text{example : type } &\text{Past\_Exports} = \{ \text{Msg\_Type out\_msg and Msg\_Type in\_msg and bag of integer index} \};
\end{align*}
\]
Appendix A. Specification of the SoFT Model

Configuration

Components may be of three different type: simple, composite, and bus. Simple components cannot be decomposed into lower level components that use the same type of interface of the simple component. Conversely, composite components are aggregations of lower level components which use the same type of interface of the composite component. Buses are associations of simple components, composite components or other buses. These types of components are described more thoroughly in appendix C. Components are defined by:

\[
\text{definition : \{component type\} \{component name\} ( \{parameters\} )}
\]

\[
\text{example : simple-component voter( )}
\]

Variables defined in the parameters definition and in the state block are global within the scope of simple components.

Command Blocks

The specification language used in this thesis utilises two loop structures. The first one is analogous to the Pascal while structure. It has the general format:

\[
\text{definition : do \{condition\} \rightarrow (block) od}
\]

\[
\text{example 1 : do } n \geq 1 \rightarrow (block) \text{ od}
\]

\[
\text{example 2 : do true } \rightarrow (block) \text{ od}
\]

In it, the block is repeated while the condition evaluates true. In the second example, the component will never exit the loop.

A second type of loop structure is used to denote that the following block will be executed for all values of a particular variable, starting from an initial value and ending at a final value:
Appendix A. Specification of the SoFT Model

**Definition:** \( fa \) (attribution) to (final value) \( \rightarrow \) (block) \( af \)

**Example:** \( fa \ i := 1 \) to \( n \) \( \rightarrow \) (block) \( af \)

Conditional statements assume the form:

\[
\text{definition} : \text{if} (\text{condition 1}) \rightarrow (\text{block 1}) \\
\quad \text{elseif} (\text{condition 2}) \rightarrow (\text{block 2}) \\
\quad \vdots \\
\quad \text{else} \rightarrow (\text{block n}) \text{ fi}
\]

**Example:**
\[
\text{if } i > 0 \rightarrow \text{(block 1)} \\
\quad \text{elseif } i = 0 \rightarrow \text{(block 2)} \\
\quad \text{else } \rightarrow \text{(block 3)} \\
\quad \text{fi}
\]

The blocks in the conditional statement are processed sequentially, as in Pascal and in C.

**Communication Primitives**

Ports can be of four types: input, output, request, and reply. The first two types are unidirectional. Messages are exported through output ports and imported through input ports. The remaining two types are bi-directional. A client writes a request to a request port and reads its reply from the same request port. A server reads a request from a reply port and writes the reply to it to the same reply port. Ports are defined by:

\[
\text{definition} : (\text{port type}) (\text{port name});
\]

**Examples:**
\[
\text{example 1} : \text{request-port } \xi; \\
\text{example 2} : \text{bag of output-port } \pi[\text{number of versions}];
\]

There are four communication primitives, namely: send, receive, request, and reply. A send primitive is used to transmit a notification message via an output port. It has the following format:
Appendix A. Specification of the SoFT Model

**Definition:**
```
definition: send (port name).(msg type)( (data) );
```

**Example 1:**
```
example 1: send ρ[2].failure( {correct msg} );
```

**Example 2:**
```
example 2: send π[1].plain( {external msg} );
```

In example 1, a message containing `{correct msg}` as data is written to port ρ[2]. This message indicates a failure. The type of the message will eventually be packed along with the data to be transmitted to the addressee. However, by stating it separately at the specification level, one can distinguish data from control. When the message is received by the addressee, the message type will be separated from the data. In example 2, an `{external msg}` is sent via port π[1]. `plain` is a reserved word that indicates that `{external msg}` may contain control information in addition to pure data. This is useful when the type of the message received by a module is unimportant to it, but is crucial to other modules to which it will forward the message.

The **receive** primitive is used by consumer and server modules to read a notification message or a request, respectively. It has the following format:

**Definition:**
```
definition: receive at (port name) (cases) within (time);
where: (cases):: (case) ⋯ (case)
and: (case):: case (msg type)( (parameters) ) → (block)
```

**Example:**
```
example: receive at ξ
  case communication_error( Communication_Status code ) →
  ⋯
  case communication_ok( Msg_Type external_msg ) →
  ⋯
  within eternity;
```

In the receive block, the thread of control goes to the end of the block when it reaches the end of the case block, as in the Pascal case. It does not enter the next case, as in the C switch.

In the example, a message is received at port ξ. Two types of messages are expected. There may be a communication error, in which case an error code will
be returned, and a successful message, in which case the data transmitted by the
sender will be received. In this example, the reserved word eternity indicates that
the caller will be blocked until a message arrives. When the caller unblocks it will
execute the code under the (case) which has been found to be true.

There is a variant form of receive that allows for the caller to read incoming
messages from various ports simultaneously. These various ports are enumerated in
a bag of ports. The caller unblocks when a message arrives at one of the ports in
the bag (the component does not block, if there is a message already waiting to be
read when the receive primitive is called). This type of receive is specified in the
following format:

\[
definition: receive \textit{at any of} \langle \textit{port bag name} \rangle \langle \textit{cases} \rangle \textit{within} \langle \textit{time} \rangle; \\
\text{where:} \langle \textit{cases}\rangle:: \langle \textit{case} \rangle \cdots \langle \textit{case} \rangle \\
\text{and:} \langle \textit{case}\rangle:: \textit{case} \langle \textit{msg type} \rangle(\langle \textit{parameters} \rangle) \rightarrow \langle \textit{block} \rangle \\
\text{example:} \textit{bag of port in := } \pi[1] + \cdots + \pi[\text{n.of.versions}]; \\
\text{receive at any of in} \\
\text{case late_msg( Msg.Type external_msg ) } \rightarrow \\
\cdots \\
\text{case communication_error( Communication.Status code ) } \rightarrow \\
\cdots \\
\text{case version_msg( Msg.Type external_msg ) } \rightarrow \\
\cdots \\
\text{within timeout;} \\
\]

In this example, the caller will unblock when a message arrives at \textit{any of} ports
\pi[1] \cdots \pi[\text{n.of.versions}].

The request primitive implements a rendez-vous. It transmits a request message
and then waits for the reply for a certain period of time. The semantics of the
request primitive, which is a combination of the semantics of the send and receive
primitives, is:
Appendix A. Specification of the SoFT Model

definition : request at (port name).(msg type)( (parameters) ) (cases) within (time);
where : (cases):: (case) · · · (case)
and : (case):: case (msg type)( (parameters) ) →(block)

example : request at α.version_msg( version_id, version_msg )
    case success( status_of, successful_msg ) →
    ...
    case indecision( ) →
    ...
    case failure( ehm_id, error_msg ) →
    ...
    within voter.timeout;

In this example, a message containing version_id, and version_msg is written to port α, stating that it is a version_msg. Although coincident, the names used in the (msg type) field and in the (parameters) field have different semantic meanings. The caller waits for a reply up to voter.timeout.

The request primitive has a variant form which allows for the caller to send the request message to various servers and then wait for their replies at several ports simultaneously. Its semantics and syntax are analogous to the receive primitive ones.

A server module receives a request from a reply port using a receive primitive. When it has processed the request, it replies to the caller with a reply call at the same reply port. A reply call has the following format:

definition : reply (port name).(msg type)( (data) );

example : reply ε.plain ( (correct msg) );

A.6.2 Syntax

This section contains the syntax of the specifications depicted in the earlier sections of this appendix and throughout the thesis. Boldface terms represent reserved words. Terms within [ and ] characters are optional. Terms in italics within ( and ) characters are defined elsewhere in the syntax definition in the format:
Appendix A. Specification of the SoFT Model

(term):: definition

Syntax Definition

(component)::

[[type declaration]]
(component declaration)

[[port declarations]]

[[state declaration]]

[[initiation]]

(main body)

[[functions]]

(type declaration):: (type definition); · · · ; (type definition);
(component declaration):: (component type) (name) (list of parameters) )
(port declarations):: (port definition); · · · ; (port definition);
(state declaration):: state (variable); · · · ; (variable);
(initiation):: initiate (statement block)
(main body):: do (condition) →(statement block) od end
(functions):: (function enumeration) · · · (function enumeration)
(function enumeration):: (function declaration): [return (type name)]

(function declaration):: function (name) (list of parameters )

(type definition):: type (name) = (type specification);
(type name):: (type) or name of a defined type
(port definition):: (absolute port definition) or (conditional port definition)
(absolute port definition):: (port type) (name)
(conditional port definition): if (condition) →(port declarations)

[ (elseif port block) · · · (elseif port block) ]

[ else →(port declarations) ] fi

(elseif port block):: elseif (condition) →(port declarations)

(name):: a literal
(number):: a numeric figure
(variable name):: a literal which labels a variable
(constant name):: a literal which labels a constant
(type specification):: (type) or (enumeration) or (structure);
(type):: integer or real or boolean or time or universal or

(enumeration name) or node or bag of (type name)

(enumeration):: {{name) or · · · or (name)}

(enumeration name):: name of an enumerated type
(structure):= (name) and · · · and (name)

(component type):: simple-component or composite-component or bus
(list of parameters):: (parameter), · · · , (parameter)

(parameter):: (constant) or (variable)
Appendix A. Specification of the SoFT Model

(constant) ::= const (name)
(variable) ::= (type) (variable name)
(port type) ::= input-port or output-port or request-port or reply-port or bag of (port type)

(statement block) ::= (statement); · · · ; (statement);
(statement) ::= (create component) or (terminate component)
(link ports) or (unlink ports)
(attribution) or (do block) or (if block) or (for all block) or
(function call) or (send block) or (receive block) or
(request block) or (reply block) or skip
(create component) ::= create (component name)((list of call parameters)
) [at (node name)];
(terminate component) ::= terminate (component name);
(link ports) ::= link [(component name).
(port name) to
[(component name).
(port name);
(unlink ports) ::= unlink [(component name).
(port name) from
[(component name).
(port name);
(component name) ::= a literal which labels a component
(port name) ::= a literal which labels a port
(attribution) ::= [(type)] (variable name) ::= (expression);
(do block) ::= do (condition) →(statement block) od
(if block) ::= if (condition) →(statement block)
[ (elseif block) · · · (elseif block) ]
[ else →(statement block) ] fi
(elseif block) ::= elseif (condition) →(statement block)
(for all block) ::= fa (variable) ::= (expression) to (expression) →(statement block) af
(send block) ::= send (port name). (msg type name)((list of call parameters) );
(receive block) ::= receive at (port enumeration) (case block) within (time);
(request block) ::= request at (port enumeration). (msg type name)((list of call parameters)
(case block) within (time);
(reply block) ::= reply (port name). (msg type name)((list of call parameters) );
(case block) ::= (case) · · · (case)
(case) ::= case (msg type) →(statement block)
(port enumeration) ::= any of (bag of port name) or (port name)
(bag of port name) ::= a literal which labels a bag of ports
(node name) ::= a literal which labels a node
(function call) ::= (function name)((list of call parameters) );
(function name) ::= a literal which labels a function
(list of call parameters) ::= (call parameter), · · · , (call parameter)
(call parameter) ::= (number) or (constant name) or (variable name) or
(attribution) or (expression)
(condition) ::= (single boolean value) or (boolean operation)
(simple boolean value) ::= true or false or (boolean variable name)
(boolean variable name) ::= a literal which labels a boolean variable
(boolean operation) ::= (condition) (boolean operand) (condition)
(boolean operand) ::= boolean operand
Appendix A. Specification of the SoFT Model

\[\text{expression} :: \text{single value} \text{ or } \text{structure value} \text{ or } \text{operation}\]
\[\text{single value} :: \text{constant name} \text{ or } \text{variable name} \text{ or } \text{number} \text{ or } \text{name} \]
\[\text{structure value} :: \{ \text{expression}, \ldots, \text{expression} \}\]
\[\text{operation} :: \text{expression} \text{ operand} \text{ expression}\]
\[\text{operand} :: \text{mathematical operand}\]
\[\text{time} :: \text{time variable} \text{ or } \text{time value}\]
\[\text{time variable} :: \text{a variable of type time}\]
\[\text{time value} :: \text{0 or eternity or number(time unit)}\]
\[\text{time unit} :: \text{milliseconds or seconds or minutes or hours or ticks}\]
\[\text{msg type} :: \text{msg type name}(\text{list of parameters})\]
\[\text{msg type name} :: \text{name} \text{ or plain}\]

Remarks

- The universal type denotes that any type is acceptable.

- The type bag of denotes a collection of elements of the same type. It is possible that the bag contains identical elements.

- Port types input-port, output-port, request-port and reply-port are defined in appendix C and in the glossary.

- Component types simple-component, composite-component, and bus are defined in appendix C.

- The time unit ticks refers to the machine clock unit.

- In the abstract representation, a text between { and } replaces a (statement block) in order to shorten the representation.

- The reserved work skip represents a void operation.
Appendix B

Complexity Evaluation of SoFT Messages

Equation 6.1 in section 6.3.2 shows that in a SoFT module in which all components have been configured, the number of messages exchanged within an interaction between an external module and the SoFT module \( N_{\text{SoFT}} \) is expressed by:

\[
N_{\text{SoFT}} = N_\ell + \sum_{i=1}^{n_\pi} N_{\pi_i} + \sum_{i=1}^{n_\rho} N_{\rho_i} + N_{\alpha_\pi} + N_{\alpha_\rho} + N_{\alpha_\alpha} + N_{\alpha_\pi} + \sum_{i=0}^{k} N_i
\]

In this appendix, this equation is evaluated for each different configuration of the SoFT model that may influence the complexity of the messages exchanged between external modules and SoFT modules (table 6.1).

In the analysis of the failure case, it is assumed that all \( k + 1 \) error handling modules will be invoked and, as a result of it, an error message will be sent to all \( n \) compensation modules (or to the versions, if they are linked to ports \( \rho_i \)). In addition, in configurations where the version outputs are assessed by the conformity test and then voted, it is assumed that the version outputs pass the conformity test. If they are rejected by the conformity test, the number of messages in the failure case is \( 2n + 1 \) less than the indicated expression, because the voter is never invoked.
Furthermore, in configurations where the version outputs are assessed by the voter first and then, if the voter reaches an agreement, such an agreement is submitted to the conformity test, it is assumed that the voter needs to assess the outputs of all the versions in order to reach an agreement. The outcome of the voting, however, does not pass the conformity test. In server configurations, the invocation of the error handling modules results in a reply to the external client indicating that an unrecoverable error has occurred. These assumptions lead to the worst possible scenario in terms of message exchange in the failure case.

B.1 Consumers

Consumer SoFT modules can only have their versions executing concurrently (section 5.2.1). The versions do not issue any output. Thus, neither the conformity test nor the voter exist.

For consumer SoFT modules without the input validation module (no error can be detected), the number of messages is evaluated as follows:

\[
N_{SoFT} = N_t + N_{\sum \pi_i} + N_{\sum \rho_i} + N_{\alpha_{is}} + N_{\alpha_{ct}} + N_{\alpha_s} + N_{\alpha_{ms}} + N_{\sum \epsilon_i}
\]

\[
= 1 + n + 0 + 0 + 0 + 0 + 0 + 0 + 0
\]

\[
= n + 1, \text{ in the success case = in the failure case}
\]

For consumer SoFT modules with the input validation module, the number of messages is evaluated as follows:

\[
N_{SoFT} = N_t + N_{\sum \pi_i} + N_{\sum \rho_i} + N_{\alpha_{is}} + N_{\alpha_{ct}} + N_{\alpha_s} + N_{\alpha_{ms}} + N_{\sum \epsilon_i}
\]
Appendix B. Complexity Evaluation of SoFT Messages

\[ N_{SoFT} = N_{x} + N_{n} + N_{a_{i}} + N_{a_{e}} + N_{a_{c}} + N_{a_{m}} + N_{a_{s}} + N_{k} \sum_{i=0}^{\infty} \]
\[ = 1 + n + 0 + 2 + 0 + 0 + 0 + 0 \\
= n + 3, \text{ in the success case} \]

\[ N_{SoFT} = N_{x} + N_{n} + N_{a_{i}} + N_{a_{e}} + N_{a_{c}} + N_{a_{m}} + N_{a_{s}} + N_{k} \sum_{i=0}^{\infty} \]
\[ = 1 + 0 + n + 2 + 0 + 0 + 0 + 2(k + 1) \\
= n + 2k + 5, \text{ in the failure case} \]

B.2 Producers

Analogously to consumer SoFT modules, producer SoFT modules can only have their versions executing concurrently (section 5.2.1). There is no input to the versions. Thus, the input validation module does not exist. The version outputs may be assessed either by the conformity test alone, or by the voter alone or by a combination of both.

For producer SoFT modules in which the version outputs are assessed only by the conformity test, the number of messages is evaluated as follows:

\[ N_{SoFT} = N_{x} + N_{n} + N_{a_{i}} + N_{a_{e}} + N_{a_{c}} + N_{a_{m}} + N_{a_{s}} + N_{k} \sum_{i=0}^{\infty} \]
\[ = 1 + n + 0 + 0 + 2 + 0 + 5(n - 1) + 0 \\
= 6n - 2, \text{ in the success case} \]

\[ N_{SoFT} = N_{x} + N_{n} + N_{a_{i}} + N_{a_{e}} + N_{a_{c}} + N_{a_{m}} + N_{a_{s}} + N_{k} \sum_{i=0}^{\infty} \]
\[ = 0 + n + n + 0 + 2n + 0 + 0 + 2(k + 1) \\
= 4n + 2k + 2, \text{ in the failure case} \]
Appendix B. Complexity Evaluation of SoFT Messages

For producer SoFT modules in which the version outputs are assessed only by the voting, the number of messages is evaluated as follows:

\[ N_{\text{SoFT}} = N_\xi + N_n \sum_{i=1}^{n_i} + N_\alpha + N_{\alpha s} + N_{\alpha c} + N_{\alpha m} + N_k \sum_{i=0}^{c_i} \]
\[ = 1 + n + 0 + 0 + 0 + 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 1 + 5 \left( n - \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 0 \]
\[ = 6n - 3 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 2, \text{ in the success case} \]

For producer SoFT modules in which the version outputs are first submitted to the conformity test and then, if accepted, submitted to the voter, the number of messages is evaluated as follows:

\[ N_{\text{SoFT}} = N_\xi + N_n \sum_{i=1}^{n_i} + N_\alpha + N_{\alpha s} + N_{\alpha c} + N_{\alpha m} + N_k \sum_{i=0}^{c_i} \]
\[ = 0 + n + 0 + 0 + 2n + 0 + 2(k + 1) \]
\[ = 4n + 2k + 2, \text{ in the failure case} \]
Appendix B. Complexity Evaluation of SoFT Messages

\[ \text{Appendix B. Complexity Evaluation of SoFT Messages} \]

For producer SoFT modules in which the version outputs are first voted and then the outcome of the voter, if successful, is submitted to the conformity test, the number of messages is evaluated as follows:

\[
N_{SoFT} = N_\xi + \sum_{i=1}^{n} N_{a_i} + \sum_{i=1}^{n} N_{\alpha_{i}} + N_{\alpha_{ct}} + N_{\alpha_{s}} + N_{\alpha_{mv}} + \sum_{i=0}^{k} N_{\delta_i} \\
= 1 + n + 0 + 0 + 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor + 1 \right) + 5 \left( n - \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 0 \\
= 6n - 3 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 4, \text{ in the success case}
\]

\[
N_{SoFT} = N_\xi + \sum_{i=1}^{n} N_{a_i} + \sum_{i=1}^{n} N_{\alpha_{i}} + N_{\alpha_{ct}} + N_{\alpha_{s}} + N_{\alpha_{mv}} + \sum_{i=0}^{k} N_{\delta_i} \\
= 0 + n + n + 0 + 2 + (2n + 1) + 0 + 2(k + 1) \\
= 4n + 2k + 5, \text{ in the failure case}
\]

B.3 Clients

Analogously to consumer and producer SoFT modules, client SoFT modules can only have their versions executing concurrently (section 5.2.1). The version requests may be assessed either by the conformity test alone, or by the voter alone or by a combination of both. The external reply can be assessed by an input validation module before it is forwarded to all the versions or, in the absence of the input validation module, it can be forwarded to all the versions directly.

For client SoFT modules in which the version outputs are assessed only
by the conformity test, and in which there is no input validation module, the number of messages is evaluated as follows:

\[
N_{\text{SoFT}} = N_e + N_{\pi} + N_{\rho} + N_{\alpha_{ei}} + N_{\alpha_{ct}} + N_{\alpha_s} + N_{\alpha_{ms}} + N_{\gamma} + \sum_{i=0}^{\infty} e_i
\]

\[
= 2 + 2n + 0 + 0 + 2 + 0 + 5(n - 1) + 0
\]

\[
= 7n - 1, \text{ in the success case}
\]

\[
N_{\text{SoFT}} = N_e + N_{\pi} + N_{\rho} + N_{\alpha_{ei}} + N_{\alpha_{ct}} + N_{\alpha_s} + N_{\alpha_{ms}} + N_{\gamma} + \sum_{i=0}^{\infty} e_i
\]

\[
= 0 + n + n + 0 + 2n + 0 + 0 + 2(k + 1)
\]

\[
= 4n + 2k + 2, \text{ in the failure case}
\]

For client SoFT modules in which the version outputs are assessed only by voting, and in which there is no input validation module, the number of messages is evaluated as follows:

\[
N_{\text{SoFT}} = N_e + N_{\pi} + N_{\rho} + N_{\alpha_{ei}} + N_{\alpha_{ct}} + N_{\alpha_s} + N_{\alpha_{ms}} + N_{\gamma} + \sum_{i=0}^{\infty} e_i
\]

\[
= 2 + 2n + 0 + 0 + 0 + [2 \left(\left\lceil \frac{n+1}{2} \right\rceil + 1\right) + 5 \left(n - \left\lceil \frac{n+1}{2} \right\rceil \right)] + 0
\]

\[
= 7n - 3 \left(\left\lceil \frac{n+1}{2} \right\rceil \right) + 3, \text{ in the success case}
\]

\[
N_{\text{SoFT}} = N_e + N_{\pi} + N_{\rho} + N_{\alpha_{ei}} + N_{\alpha_{ct}} + N_{\alpha_s} + N_{\alpha_{ms}} + N_{\gamma} + \sum_{i=0}^{\infty} e_i
\]

\[
= 0 + n + n + 0 + 0 + (2n + 1) + 0 + 2(k + 1)
\]

\[
= 4n + 2k + 3, \text{ in the failure case}
\]
Appendix B. Complexity Evaluation of SoFT Messages

For client SoFT modules in which the version outputs are first submitted to the conformity test and then, if accepted, submitted to the voter, and in which there is no input validation module, the number of messages is evaluated as follows:

\[
N_{\text{SoFT}} = N_\ell + \sum_{i=1}^{\pi_i} N_{\alpha_i} + \sum_{i=1}^{\rho_i} N_{\alpha_{ct}} + N_{\alpha_s} + N_{\alpha_{m_s}} + \sum_{i=0}^{\varepsilon_i} N_{\delta}
\]

\[
= 2 + 2n + 0 + 0 + 2 \left( \left\lceil \frac{n+1}{2} \right\rceil \right) + \left[ \left\lceil \frac{n+1}{2} \right\rceil \right] + 1 + \left( n - \left\lceil \frac{n+1}{2} \right\rceil \right) + 0
\]

\[
= 7n - \left( \left\lceil \frac{n+1}{2} \right\rceil \right) + 3, \text{ in the success case}
\]

For client SoFT modules in which the version outputs are first voted and then the outcome of the voter, if successful, is submitted to the conformity test, and in which there is no input validation module, the number of messages is evaluated as follows:

\[
N_{\text{SoFT}} = N_\ell + \sum_{i=1}^{\pi_i} N_{\alpha_i} + \sum_{i=1}^{\rho_i} N_{\alpha_{ct}} + N_{\alpha_s} + N_{\alpha_{m_s}} + \sum_{i=0}^{\varepsilon_i} N_{\delta}
\]

\[
= 0 + n + n + 0 + 2n + (2n + 1) + 0 + 2(k + 1)
\]

\[
= 6n + 2k + 3, \text{ in the failure case}
\]

For client SoFT modules in which the version outputs are first submitted to the conformity test and then, if accepted, submitted to the voter, and in which there is no input validation module, the number of messages is evaluated as follows:

\[
N_{\text{SoFT}} = N_\ell + \sum_{i=1}^{\pi_i} N_{\alpha_i} + \sum_{i=1}^{\rho_i} N_{\alpha_{ct}} + N_{\alpha_s} + N_{\alpha_{m_s}} + \sum_{i=0}^{\varepsilon_i} N_{\delta}
\]

\[
= 2 + 2n + 0 + 0 + 2 + \left[ \left\lceil \frac{n+1}{2} \right\rceil \right] + 5 \left( n - \left\lceil \frac{n+1}{2} \right\rceil \right) + 0
\]

\[
= 7n - 3 \left( \left\lceil \frac{n+1}{2} \right\rceil \right) + 5, \text{ in the success case}
\]
Appendix B. Complexity Evaluation of SoFT Messages

\[ N_{\text{SoFT}} = N_\ell + \sum_{i=1}^{\ell_i} N_\ell + \sum_{i=1}^{\ell_i} N_{\phi_i} + N_{\alpha_is} + N_{\alpha_cl} + N_{\alpha_s} + N_{\beta_is} + N_{\beta_c} \]
\[ = 0 + n + n + 0 + 2 + (2n + 1) + 0 + 2(k + 1) \]
\[ = 4n + 2k + 5, \text{ in the failure case} \]

For client SoFT modules in which the version outputs are assessed only by the conformity test, and in which the input validation module exists, the number of messages is evaluated as follows:

\[ N_{\text{SoFT}} = N_\ell + \sum_{i=1}^{\ell_i} N_\ell + \sum_{i=1}^{\ell_i} N_{\phi_i} + N_{\alpha_is} + N_{\alpha_cl} + N_{\alpha_s} + N_{\beta_is} + N_{\beta_c} \]
\[ = 0 + n + n + 0 + 2 + 2 + 0 + 5(n - 1) + 0 \]
\[ = 7n + 1, \text{ in the success case} \]

\[ N_{\text{SoFT}} = N_\ell + \sum_{i=1}^{\ell_i} N_\ell + \sum_{i=1}^{\ell_i} N_{\phi_i} + N_{\alpha_is} + N_{\alpha_cl} + N_{\alpha_s} + N_{\beta_is} + N_{\beta_c} \]
\[ = 0 + n + n + 0 + 2n + 0 + 0 + 2(k + 1) \]
\[ = 4n + 2k + 2, \text{ in the failure case} \]

For client SoFT modules in which the version outputs are assessed only by voting, and in which the input validation module exists, the number of messages is evaluated as follows:

\[ N_{\text{SoFT}} = N_\ell + \sum_{i=1}^{\ell_i} N_\ell + \sum_{i=1}^{\ell_i} N_{\phi_i} + N_{\alpha_is} + N_{\alpha_cl} + N_{\alpha_s} + N_{\beta_is} + N_{\beta_c} \]
\[ = 2 + 2n + 0 + 2 + 0 + 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 5 \left( n - \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 0 \]
\[ = 7n - 3 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 5, \text{ in the success case} \]
Appendix B. Complexity Evaluation of SoFT Messages

\[ N_{SoFT} = \sum_{i=1}^{\pi_i} N_{\pi_i} + \sum_{i=1}^{\rho_i} N_{\rho_i} + N_{\text{act}} + N_{\text{act}} + N_{\text{act}} + N_{\text{act}} + N_{\text{act}} + N_{k} \sum_{i=0}^{c_i} \]

\[ = 0 + n + n + 0 + 0 + (2n + 1) + 0 + 2(k + 1) \]

\[ = 4n + 2k + 3, \text{ in the failure case} \]

For client SoFT modules in which the version outputs are first submitted to the conformity test and then, if accepted, submitted to the voter, and in which the input validation module exists, the number of messages is evaluated as follows:

\[ N_{SoFT} = \sum_{i=1}^{\pi_i} N_{\pi_i} + \sum_{i=1}^{\rho_i} N_{\rho_i} + N_{\text{act}} + N_{\text{act}} + N_{\text{act}} + N_{\text{act}} + N_{\text{act}} + N_{k} \sum_{i=0}^{c_i} \]

\[ = 2 + 2n + 0 + 2 + 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 1 + \]

\[ + 5 \left( n - \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 0 \]

\[ = 7n - \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 5, \text{ in the success case} \]

\[ N_{SoFT} = \sum_{i=1}^{\pi_i} N_{\pi_i} + \sum_{i=1}^{\rho_i} N_{\rho_i} + N_{\text{act}} + N_{\text{act}} + N_{\text{act}} + N_{\text{act}} + N_{\text{act}} + N_{k} \sum_{i=0}^{c_i} \]

\[ = 0 + n + n + 0 + 2n + (2n + 1) + 0 + 2(k + 1) \]

\[ = 6n + 2k + 3, \text{ in the failure case} \]

For client SoFT modules in which the version outputs are first voted and then the outcome of the voter, if successful, is submitted to the conformity test, and in which the input validation module exists, the number of messages is evaluated as follows:
Appendix B. Complexity Evaluation of SoFT Messages

\[ N_{SoFT} = N_\xi + N_{\alpha v} + \sum_{i=1}^{n} \pi_i + N_{\alpha v e} + N_{\alpha v t} + N_{\alpha s} + N_{\alpha m e} + N_{k} \]

\[ = 2 + 2n + 0 + 2 + 2 + \left[ 2 \left( \left\lceil \frac{n+1}{2} \right\rceil \right) + 1 \right] + 5 \left( n - \left\lceil \frac{n+1}{2} \right\rceil \right) + 0 \]

\[ = 7n - 3 \left( \left\lceil \frac{n+1}{2} \right\rceil \right) + 7, \text{ in the success case} \]

\[ N_{SoFT} = N_\xi + N_{\alpha v} + \sum_{i=1}^{n} \pi_i + N_{\alpha v e} + N_{\alpha v t} + N_{\alpha s} + N_{\alpha m e} + N_{k} \]

\[ = 0 + n + n + 0 + 2 + (2n + 1) + 0 + 2(k + 1) \]

\[ = 4n + 2k + 5, \text{ in the failure case} \]

B.4 Servers

Contrary to consumer, producer, and client SoFT modules, server SoFT modules can have their versions executing either sequentially or concurrently (section 5.2.1). The external request can be assessed by an input validation module before it is forwarded to the versions or, in the absence of the input validation module, it can be forwarded to the versions directly. The version replies may be assessed either by the conformity test alone, or by the voter alone or by a combination of both.

B.4.1 Concurrent Servers

For server SoFT modules in which the versions execute concurrently, the version outputs are assessed only by the conformity test, and in which there is no input validation module, the number of messages is evaluated as follows:
Appendix B. Complexity Evaluation of SoFT Messages

\[ N_{\text{SoFT}} = \sum_{i=1}^{\pi_i} + N_{\text{ai}^i} + N_{\alpha_{ai}^i} + N_{\alpha_{ai}^e} + N_{\alpha_{ai}^ct} + N_{\alpha_{ai}^*} + N_{\alpha_{ai}^m} + N_{\sum_{i=0}^{e_i}} \]

\[ = 2 + 2n + 0 + 0 + 2 + 0 + 5(n - 1) + 0 \]

\[ = 7n - 1, \text{ in the success case} \]

\[ N_{\text{SoFT}} = \sum_{i=1}^{\pi_i} + N_{\text{ai}^i} + N_{\alpha_{ai}^i} + N_{\alpha_{ai}^ct} + N_{\alpha_{ai}^*} + N_{\alpha_{ai}^m} + N_{\sum_{i=0}^{e_i}} \]

\[ = 2 + 2n + n + 0 + 2n + 0 + 0 + 2(k + 1) \]

\[ = 5n + 2k + 4, \text{ in the failure case} \]

For server SoFT modules in which the versions execute concurrently, the version outputs are assessed only by voting, and in which there is no input validation module, the number of messages is evaluated as follows:

\[ N_{\text{SoFT}} = \sum_{i=1}^{\pi_i} + N_{\text{ai}^i} + N_{\alpha_{ai}^i} + N_{\alpha_{ai}^ct} + N_{\alpha_{ai}^*} + N_{\alpha_{ai}^m} + N_{\sum_{i=0}^{e_i}} \]

\[ = 2 + 2n + 0 + 0 + 0 + \left[ 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor + 1 \right) + 5 \left( n - \left\lfloor \frac{n+1}{2} \right\rfloor \right) \right] + 0 \]

\[ = 7n - 3 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 3, \text{ in the success case} \]

\[ N_{\text{SoFT}} = \sum_{i=1}^{\pi_i} + N_{\text{ai}^i} + N_{\alpha_{ai}^i} + N_{\alpha_{ai}^ct} + N_{\alpha_{ai}^*} + N_{\alpha_{ai}^m} + N_{\sum_{i=0}^{e_i}} \]

\[ = 2 + 2n + n + 0 + 0 + (2n + 1) + 0 + 2(k + 1) \]

\[ = 5n + 2k + 5, \text{ in the failure case} \]

For server SoFT modules in which the versions execute concurrently, the version outputs are first submitted to the conformity test and then, if
Appendix B. Complexity Evaluation of SoFT Messages

accepted, submitted to the voter, and in which there is no input validation module, the number of messages is evaluated as follows:

\[ N_{\text{SoFT}} = N_\ell + N_\sum_{i=1}^n \pi_i + N_\sum_{i=1}^n \rho_i + N_{\alpha_{iv}} + N_{\alpha_{vt}} + N_{\alpha_{v}} + N_{\alpha_{mv}} + N_k \sum_{i=0}^c_i \]

\[ = 2 + 2n + 0 + 0 + 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 1 + \\
+ 5 \left( n - \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 0 \]

\[ = 7n - \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 3, \text{ in the success case} \]

\[ N_{\text{SoFT}} = N_\ell + N_\sum_{i=1}^n \pi_i + N_\sum_{i=1}^n \rho_i + N_{\alpha_{iv}} + N_{\alpha_{vt}} + N_{\alpha_{v}} + N_{\alpha_{mv}} + N_k \sum_{i=0}^c_i \]

\[ = 2 + 2n + n + 0 + 2n + (2n+1) + 0 + 2(k+1) \]

\[ = 7n + 2k + 5, \text{ in the failure case} \]

For server SoFT modules in which the versions execute concurrently, the version outputs are first voted and then the outcome of the voter, if successful, is submitted to the conformity test, and in which there is no input validation module, the number of messages is evaluated as follows:

\[ N_{\text{SoFT}} = N_\ell + N_\sum_{i=1}^n \pi_i + N_\sum_{i=1}^n \rho_i + N_{\alpha_{iv}} + N_{\alpha_{vt}} + N_{\alpha_{v}} + N_{\alpha_{mv}} + N_k \sum_{i=0}^c_i \]

\[ = 2 + 2n + 0 + 0 + 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 1 + 5 \left( n - \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 0 \]

\[ = 7n - 3 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 5, \text{ in the success case} \]

\[ N_{\text{SoFT}} = N_\ell + N_\sum_{i=1}^n \pi_i + N_\sum_{i=1}^n \rho_i + N_{\alpha_{iv}} + N_{\alpha_{vt}} + N_{\alpha_{v}} + N_{\alpha_{mv}} + N_k \sum_{i=0}^c_i \]
Appendix B. Complexity Evaluation of SoFT Messages

\[ n_{\text{SoFT}} = n_0 + n_\alpha + n_{\alpha_{\text{st}}} + n_{\alpha_{\text{ct}}} + n_{\alpha_{\text{sv}}} + n_{\alpha_{\text{ms}}} + n_k. \]

\[ = 2 + 2n + n + 0 + 2 + 2(n + 1) + 0 + 2(k + 1) \]

\[ = 5n + 2k + 7, \text{ in the failure case} \]

For server SoFT modules in which the versions execute concurrently, the version outputs are assessed only by the conformity test, and in which the input validation module exists, the number of messages is evaluated as follows:

\[ n_{\text{SoFT}} = n_0 + n_\alpha + n_{\alpha_{\text{st}}} + n_{\alpha_{\text{ct}}} + n_{\alpha_{\text{sv}}} + n_{\alpha_{\text{ms}}} + n_k. \]

\[ = 2 + 2n + n + 2 + 2n + 0 + 0 + 5(n - 1) + 0 \]

\[ = 7n + 1, \text{ in the success case} \]

\[ n_{\text{SoFT}} = n_0 + n_\alpha + n_{\alpha_{\text{st}}} + n_{\alpha_{\text{ct}}} + n_{\alpha_{\text{sv}}} + n_{\alpha_{\text{ms}}} + n_k. \]

\[ = 2 + 2n + n + 2 + 2n + 0 + 0 + 2(k + 1) \]

\[ = 5n + 2k + 6, \text{ in the failure case} \]

For server SoFT modules in which the versions execute concurrently, the version outputs are assessed only by voting, and in which the input validation module exists, the number of messages is evaluated as follows:

\[ n_{\text{SoFT}} = n_0 + n_\alpha + n_{\alpha_{\text{st}}} + n_{\alpha_{\text{ct}}} + n_{\alpha_{\text{sv}}} + n_{\alpha_{\text{ms}}} + n_k. \]

\[ = 2 + 2n + 0 + 2 + 2 + 2 \left( \left\lceil \frac{n + 1}{2} \right\rceil \right) + 1 + 5 \left( n - \left\lceil \frac{n + 1}{2} \right\rceil \right) + 0 \]

\[ = 7n - 3 \left( \left\lceil \frac{n + 1}{2} \right\rceil \right) + 5, \text{ in the success case} \]
Appendix B. Complexity Evaluation of SoFT Messages

\[ N_{\text{SoFT}} = N_\xi + N_{\sum_{i=1}^{\pi_i}} + N_{\sum_{i=1}^{\rho_i}} + N_{\alpha_{es}} + N_{\alpha_{ct}} + N_{\alpha_{s}} + N_{\alpha_{ms}} + N_{\sum_{i=0}^{k}} \]

\[ = 2 + 2n + n + 2 + 0 + (2n + 1) + 0 + 2(k + 1) \]

\[ = 5n + 2k + 7, \text{ in the failure case} \]

For server SoFT modules in which the versions execute concurrently, the version outputs are first submitted to the conformity test and then, if accepted, submitted to the voter, and in which the input validation module exists, the number of messages is evaluated as follows:

\[ N_{\text{SoFT}} = N_\xi + N_{\sum_{i=1}^{\pi_i}} + N_{\sum_{i=1}^{\rho_i}} + N_{\alpha_{es}} + N_{\alpha_{ct}} + N_{\alpha_{s}} + N_{\alpha_{ms}} + N_{\sum_{i=0}^{k}} \]

\[ = 2 + 2n + 0 + 2 + 2 \left( \left[ \frac{n+1}{2} \right] \right) + 2 \left( \left[ \frac{n+1}{2} \right] + 1 \right) + \]

\[ + 5 \left( n - \left[ \frac{n+1}{2} \right] \right) + 0 \]

\[ = 7n - \left( \left[ \frac{n+1}{2} \right] \right) + 5, \text{ in the success case} \]

\[ N_{\text{SoFT}} = N_\xi + N_{\sum_{i=1}^{\pi_i}} + N_{\sum_{i=1}^{\rho_i}} + N_{\alpha_{es}} + N_{\alpha_{ct}} + N_{\alpha_{s}} + N_{\alpha_{ms}} + N_{\sum_{i=0}^{k}} \]

\[ = 2 + 2n + n + 2 + 2n + (2n + 1) + 0 + 2(k + 1) \]

\[ = 7n + 2k + 7, \text{ in the failure case} \]

For server SoFT modules in which the versions execute concurrently, the version outputs are first voted and then the outcome of the voter, if successful, is submitted to the conformity test, and in which the input validation module exists, the number of messages is evaluated as follows:
Appendix B. Complexity Evaluation of SoFT Messages

\[ \mathcal{N}_{\text{SoFT}} = N_t + \sum_{i=1}^{\xi_i} + \sum_{i=1}^{\rho_i} + N_{\alpha_{\epsilon_i}} + N_{\alpha_{\epsilon_1}} + N_{\alpha_{\epsilon_2}} + N_{\alpha_{\epsilon_3}} + N_{\alpha_{\epsilon_4}} + N_{\alpha_{\epsilon_5}} + N_{\alpha_{\epsilon_6}} + N_{\alpha_{\epsilon_7}} + N_{\alpha_{\epsilon_8}} + N_{\alpha_{\epsilon_9}} + N_{\alpha_{\epsilon_{10}}} \]

\[ = 2 + 2n + 0 + 2 + 2 + \left[ 2 \left( \left\lceil \frac{n+1}{2} \right\rceil \right) + 1 \right] + 5 \left( n - \left\lceil \frac{n+1}{2} \right\rceil \right) + 0 \]

\[ = 7n - 3 \left( \left\lceil \frac{n+1}{2} \right\rceil \right) + 7, \text{ in the success case} \]

\[ \mathcal{N}_{\text{SoFT}} = N_t + \sum_{i=1}^{\xi_i} + \sum_{i=1}^{\rho_i} + N_{\alpha_{\epsilon_i}} + N_{\alpha_{\epsilon_1}} + N_{\alpha_{\epsilon_2}} + N_{\alpha_{\epsilon_3}} + N_{\alpha_{\epsilon_4}} + N_{\alpha_{\epsilon_5}} + N_{\alpha_{\epsilon_6}} + N_{\alpha_{\epsilon_7}} + N_{\alpha_{\epsilon_8}} + N_{\alpha_{\epsilon_9}} + N_{\alpha_{\epsilon_{10}}} \]

\[ = 2 + 2n + n + 2 + 2 + (2n + 1) + 0 + 2(k + 1) \]

\[ = 5n + 2k + 9, \text{ in the failure case} \]

B.4.2 Sequential Servers

For server SoFT modules in which the versions execute sequentially, the version outputs are assessed only by the conformity test, and in which there is no input validation module, the number of messages is evaluated as follows:

\[ \mathcal{N}_{\text{SoFT}} = N_t + \sum_{i=1}^{\xi_i} + \sum_{i=1}^{\rho_i} + N_{\alpha_{\epsilon_i}} + N_{\alpha_{\epsilon_1}} + N_{\alpha_{\epsilon_2}} + N_{\alpha_{\epsilon_3}} + N_{\alpha_{\epsilon_4}} + N_{\alpha_{\epsilon_5}} + N_{\alpha_{\epsilon_6}} + N_{\alpha_{\epsilon_7}} + N_{\alpha_{\epsilon_8}} + N_{\alpha_{\epsilon_9}} + N_{\alpha_{\epsilon_{10}}} \]

\[ = 2 + 2 + 0 + 0 + 2 + 0 + 0 \]

\[ = 6, \text{ in the success case} \]

\[ \mathcal{N}_{\text{SoFT}} = N_t + \sum_{i=1}^{\xi_i} + \sum_{i=1}^{\rho_i} + N_{\alpha_{\epsilon_i}} + N_{\alpha_{\epsilon_1}} + N_{\alpha_{\epsilon_2}} + N_{\alpha_{\epsilon_3}} + N_{\alpha_{\epsilon_4}} + N_{\alpha_{\epsilon_5}} + N_{\alpha_{\epsilon_6}} + N_{\alpha_{\epsilon_7}} + N_{\alpha_{\epsilon_8}} + N_{\alpha_{\epsilon_9}} + N_{\alpha_{\epsilon_{10}}} \]

\[ = 2 + 2n + n + 2 + 2n + 0 + 0 + 2(k + 1) \]

\[ = 5n + 2k + 4, \text{ in the failure case} \]
For server SoFT modules in which the versions execute sequentially, the version outputs are assessed only by voting, and in which there is no input validation module, the number of messages is evaluated as follows:

\[
N_{\text{SoFT}} = N_t + N_{a_i} \sum_{i=1}^{n} \pi_i + N_{a_o} \sum_{i=1}^{\rho_i} + N_{a_{is}} + N_{a_{ct}} + N_{a_{s}} + N_{a_{ms}} + N_k \sum_{i=0}^{e_i}
\]

\[
= 2 + 2 \left( \left\lceil \frac{n+1}{2} \right\rceil \right) + 0 + 0 + 0 + 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor + 1 \right) + 0 + 0
\]

\[
= 4 \left( \left\lceil \frac{n+1}{2} \right\rceil \right) + 3, \text{ in the success case}
\]

For server SoFT modules in which the versions execute sequentially, the version outputs are first submitted to the conformity test and then, if accepted, submitted to the voter, and in which there is no input validation module, the number of messages is evaluated as follows:

\[
N_{\text{SoFT}} = N_t + N_{a_i} \sum_{i=1}^{n} \pi_i + N_{a_o} \sum_{i=1}^{\rho_i} + N_{a_{is}} + N_{a_{ct}} + N_{a_{s}} + N_{a_{ms}} + N_k \sum_{i=0}^{e_i}
\]

\[
= 2 + 2n + n + 0 + 0 + (2n + 1) + 0 + 2(k + 1)
\]

\[
= 5n + 2k + 5, \text{ in the failure case}
\]
Appendix B. Complexity Evaluation of SoFT Messages

\[ N_{\text{SoFT}} = N_t + \sum_{i=1}^{\pi_i} + N_s + N_{\alpha_{iv}} + N_{\alpha_{ct}} + N_{\alpha_s} + N_{\alpha_{ms}} + N_{\beta} \sum_{i=0}^{c_i} \]

\[ = 2 + 2n + n + 0 + 2n + (2n + 1) + 0 + 2(k + 1) \]
\[ = 7n + 2k + 5, \text{ in the failure case} \]

For server SoFT modules in which the versions execute sequentially, the version outputs are first voted and then the outcome of the voter, if successful, is submitted to the conformity test, and in which there is no input validation module, the number of messages is evaluated as follows:

\[ N_{\text{SoFT}} = N_t + \sum_{i=1}^{\pi_i} + N_s + N_{\alpha_{iv}} + N_{\alpha_{ct}} + N_{\alpha_s} + N_{\alpha_{ms}} + N_{\beta} \sum_{i=0}^{c_i} \]

\[ = 2 + 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 0 + 2 + \left( 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 1 \right) + 0 + 0 \]
\[ = 4 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 5, \text{ in the success case} \]

\[ N_{\text{SoFT}} = N_t + \sum_{i=1}^{\pi_i} + N_s + N_{\alpha_{iv}} + N_{\alpha_{ct}} + N_{\alpha_s} + N_{\alpha_{ms}} + N_{\beta} \sum_{i=0}^{c_i} \]

\[ = 2 + 2n + n + 0 + 2 + (2n + 1) + 0 + 2(k + 1) \]
\[ = 5n + 2k + 7, \text{ in the failure case} \]

For server SoFT modules in which the versions execute sequentially, the version outputs are assessed only by the conformity test, and in which the input validation module exists, the number of messages is evaluated as follows:

\[ N_{\text{SoFT}} = N_t + \sum_{i=1}^{\pi_i} + N_s + N_{\alpha_{iv}} + N_{\alpha_{ct}} + N_{\alpha_s} + N_{\alpha_{ms}} + N_{\beta} \sum_{i=0}^{c_i} \]
Appendix B. Complexity Evaluation of SoFT Messages

\[ N_{SoFT} = N_\varepsilon + N_{a_0} + N_{a_1} + N_{a_{ct}} + N_{a_r} + N_{a_{mv}} + N_k \sum_{i=0}^k \varepsilon_i \]

\[ = 2 + 2 + 0 + 2 + 0 + 0 + 0 \]

\[ = 8, \text{ in the success case} \]

\[ \sum_{i=1}^n \pi_i^t + \sum_{i=1}^n \rho_i + N_{a_{ct}} + N_{a_r} + N_{a_{mv}} + N_k \sum_{i=0}^k \varepsilon_i \]

\[ = 2 + 2n + n + 2 + 2n + 0 + 0 + 2(k + 1) \]

\[ = 5n + 2k + 6, \text{ in the failure case} \]

For server SoFT modules in which the versions execute sequentially, the version outputs are assessed only by voting, and in which the input validation module exists, the number of messages is evaluated as follows:

\[ N_{SoFT} = N_\varepsilon + N_{a_0} + N_{a_1} + N_{a_{ct}} + N_{a_r} + N_{a_{mv}} + N_k \sum_{i=0}^k \varepsilon_i \]

\[ = 2 + 2 \left( \frac{n + 1}{2} \right) + 0 + 2 + 0 + \left[ 2 \left( \left\lfloor \frac{n + 1}{2} \right\rfloor \right) + 1 \right] + 0 + 0 \]

\[ = 4 \left( \left\lfloor \frac{n + 1}{2} \right\rfloor \right) + 5, \text{ in the success case} \]

\[ = 2 + 2n + n + 2 + 0 + (2n + 1) + 0 + 2(k + 1) \]

\[ = 5n + 2k + 7, \text{ in the failure case} \]

For server SoFT modules in which the versions execute sequentially, the version outputs are first submitted to the conformity test and then, if accepted, submitted to the voter, and in which the input validation module exists, the number of messages is evaluated as follows:
Appendix B. Complexity Evaluation of SoFT Messages

\[ N_{\text{SoFT}} = N_\xi + \sum_{i=1}^{n} N_{x_i} + \sum_{i=1}^{\rho_i} N_{\alpha_{iv}} + N_{\alpha_{iv}} + N_{\alpha_{ct}} + N_{\alpha_{ct}} + N_{\alpha_{mv}} + N_{\alpha_{mv}} + N_{\sum_{i=0}^{c_i}} \]

\[ = 2 + 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 0 + 2 + 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + \]

\[ + \left[ 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 1 \right] + 0 + 0 \]

\[ = 6 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 5, \text{ in the success case} \]

\[ N_{\text{SoFT}} = N_\xi + \sum_{i=1}^{n} N_{x_i} + \sum_{i=1}^{\rho_i} N_{\alpha_{iv}} + N_{\alpha_{iv}} + N_{\alpha_{ct}} + N_{\alpha_{ct}} + N_{\alpha_{mv}} + N_{\alpha_{mv}} + N_{\sum_{i=0}^{c_i}} \]

\[ = 2 + 2n + n + 0 + 2n + (2n + 1) + 2 + 2(k + 1) \]

\[ = 7n + 2k + 7, \text{ in the failure case} \]

For server SoFT modules in which the versions execute sequentially, the version outputs are first voted and then the outcome of the voter, if successful, is submitted to the conformity test, and in which the input validation module exists, the number of messages is evaluated as follows:

\[ N_{\text{SoFT}} = N_\xi + \sum_{i=1}^{n} N_{x_i} + \sum_{i=1}^{\rho_i} N_{\alpha_{iv}} + N_{\alpha_{iv}} + N_{\alpha_{ct}} + N_{\alpha_{ct}} + N_{\alpha_{mv}} + N_{\alpha_{mv}} + N_{\sum_{i=0}^{c_i}} \]

\[ = 2 + 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 0 + 2 + 2 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 1 + 0 + 0 \]

\[ = 4 \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + 7, \text{ in the success case} \]

\[ N_{\text{SoFT}} = N_\xi + \sum_{i=1}^{n} N_{x_i} + \sum_{i=1}^{\rho_i} N_{\alpha_{iv}} + N_{\alpha_{iv}} + N_{\alpha_{ct}} + N_{\alpha_{ct}} + N_{\alpha_{mv}} + N_{\alpha_{mv}} + N_{\sum_{i=0}^{c_i}} \]

\[ = 2 + 2n + n + 0 + 2 + (2n + 1) + 2 + 2(k + 1) \]

\[ = 5n + 2k + 9, \text{ in the failure case} \]
Appendix C

POPE: a Port-Oriented Programming Environment

POPE is a port-oriented programming environment used to design and implement software fault-tolerant modules based on the SoFT framework. POPE, however, is by no means restricted to fault-tolerant applications. POPE is based on the programming-in-the-small versus programming-in-the-large approach [DK76]. Programming-in-the-small versus programming-in-the-large has been successfully used in the past for investigating selective software fault-tolerance in distributed systems [Loq84, Ani89]. This approach is suitable for supporting transparent and selective software fault-tolerance, because it allows for diversely designed versions, assessment components, and other components of the SoFT model to be developed and tested independently and assembled at a later stage.

C.1 Primitive Components

Computing systems are made of several components working under the control of a design (section 2.1). Programming-in-the-small requires modules to be designed, programmed, implemented, and tested independently. Modules address one another indirectly by means of local named communication interfaces called ports.
Programming-in-the-large is accomplished by linking ports accordingly.

Processes communicate by exchanging notification, request, and reply messages. Notification messages carry information that does not need to be responded to. Conversely, a request message requires a reply from the recipient. There are four different types of ports in POPE, namely: input, output, request, and reply. Notification messages are written to output ports and read from input ports. Request messages are sent through request ports. A request message is received at a reply port. The reply to the request is issued at the same port and is delivered to the request port at which the request originated. The reply is not delivered to all request ports linked to the reply port, because there is no guarantee that it will be of any use to components other than the sender of the request.

Concurrent processes need frequently to read and update information that is shared amongst them. However, unco-ordinated update of shared data in concurrent systems can lead to inconsistency and loss of information. Amongst the various methods of providing mutual exclusion [MD74, HGLS78, Dei84], encapsulation [Cox86, Mey88] has the advantage of hiding the internal data structure from other components. In addition, concurrency control is automatically attained as the communication system imposes an order on user requests by placing them in the message queue of the process which encapsulates the information. Thus, primitive components in POPE are processes that encapsulate data and communicate via ports.

Primitive components should not link their own ports, because this would be a disguised form of direct addressing. Hence, ports are to be linked by configuration processes. Output ports can only be linked to input ports. Analogously, configuration processes are able to link request ports only to reply ports. Configuration processes can link and unlink ports of processes executing locally and at remote sites. The minimum number of configuration processes in a system is one. However, there may exist as many configuration processes as the user finds appropriate.
Appendix C. POPE: a Port-Oriented Programming Environment

Primitive components can be seen as "software chips" that have their "pins" logically linked to others to form more complex "circuits". Figure C.1.a illustrates the notation used in this dissertation to represent components in POPE. Figure C.1.b depicts three interacting components. Component A sends notification messages through port $a_{out}$. These messages are received by component $B$ at port $b_{in}$. Component $B$ communicates with component $C$ by issuing requests and receiving replies at port $b_{req}$. The requests issued by $B$ are delivered at port $c_{rep}$. $C$ replies at the same port and sends notification messages at port $c_{out}$, which are received by $A$ at $a_{in}$.

C.2 The Kernel

Configuration and inter-process communication are performed by a distributed kernel. The distributed kernel consists of interacting local kernel modules situated at each of the nodes of the network. Kernel functions are implemented by a set of primitives. Application and configuration components use a library of functions, which is linked to them, to call the kernel primitives. Calls are received and completed by the local kernel if they refer to processes executing on that site. Otherwise, the
local kernel interacts with remote kernel modules. Configuration processes can act on both local and remote components.

The kernel is transparent to the user. From the user's point of view, messages are copied directly from the exporter to the importer. However, configuration and communication primitives are actually executed by the kernel. Communication primitives have the same syntax regardless of the location of the recipient.

Figure C.2 depicts a simple client-server configuration. Client component $C$, situated at machine $\alpha$, and server component $S$, executing at machine $\beta$, are configured by component $Conf$, located at node $\alpha$. Kernel calls made by $Conf$ are received by the local kernel at machine $\alpha$. However, as Server $S$ and configuration component $Conf$ are at different nodes, the local kernel communicates with the kernel at machine $\beta$ in order to complete the execution of configuration primitives. Messages written by $C$ to port $c_{out}$ are firstly transferred to the kernel module at machine $\alpha$. When the link between port $c_{out}$ and port $s_{in}$ has been confirmed, the message is sent to the kernel module at machine $\beta$, which finally places it in the message queue of port $s_{in}$. If process $S$ has been waiting for a message at $s_{in}$, it will be notified of its arrival by the local kernel. In the diagram, dashed lines denote the actual data flow, whereas the full line indicates the logical path of messages, from the user's point of view.

C.3 Cluster Components

The number of modules and links that are necessary to build large distributed systems directly from primitive components make configuration and management virtually impossible to handle. Furthermore, designing a complex system requires different levels of abstraction. In large distributed applications, primitive components which collaborate to perform a higher level function can be grouped, forming cluster components.
There are three situations where it is convenient to work with cluster components rather than with primitive components [CZ85]. The first of them consists of services performed by a set of interacting components that can be regarded by an external viewer as being produced by a single element. Another example of cluster components arises in pipeline configurations, where the output of one process is applied as input to its successor. Any job control operation performed on one of them affects the others. There is no reason for suspending, resuming, or terminating one of them without doing the same to the others. Finally, there is the case of client components accessing multiple servers. It is much simpler for the clients to deal with a single abstract server than with different components which perform analogous functions. Besides, abstract servers have the appeal of hiding the identity and location of the server that actually executes the operation.

Cluster components can be aggregations or associations of lower level components [PT91]. *Aggregation* is a form of abstraction in which interacting components are regarded as one higher level component. The internal structure of an aggrega-
tion is invisible from outside and, therefore, it is impossible for an external element to have direct access to its members. **Association** is a form of abstraction in which components are regarded as a single higher level set of components. Associated components do not have to interact necessarily. Members of an association can be accessed by external elements directly and can belong to more than one association at a time. Interacting and pipeline components fall into the aggregation class, whereas the abstract server configuration is best represented as an association of individual servers. Aggregations are implemented in POPE as **composite components** and associations as **communication buses**.

### C.3.1 Composite Components

Composite components are aggregations of dependent (or encapsulated) elements. Thus, members of a composite component cannot join more than one composite component at a time, nor can their ports be directly linked to components that do not belong to the composite component. Composite components communicate with other modules via ports. Composite components can be made of primitive components or of other composite components.

The first step towards configuring a composite component is to create its members. Member ports are then linked amongst each other, according to the design of the composite component. Finally, ports to which external messages will be delivered and those from which internal messages will be exported are linked to their respective composite component ports. Link and unlink primitive calls have the same syntax both for ports belonging to primitive components and for those which are part of composite components.

An example of composite component is depicted in figure C.3. Composite component \( \Gamma \) has two input ports, namely \( p_{T_1} \) and \( p_{T_2} \), and one output port, \( p_{T_3} \). External modules may be connected to ports \( p_{T_1}, p_{T_2}, \) and \( p_{T_3} \), but cannot be linked to any of the ports of the member components. Components \( A, B, C, D, \) and \( E \) communicate
amongst each other in order to provide the service known to other modules in the system as $\Gamma$. Messages written to external ports linked to $p_{T_1}$ will be delivered at $p_{A_1}$ and those sent to port $p_{T_3}$ will be received by process $B$. Composite component $\Gamma$ exports its outputs, which have actually been produced by modules $D$ and $E$, at port $p_{T_3}$. The figure shows part of another composite component, named $\Delta$, that is linked to $\Gamma$ via port $p_{A_1}$. Messages exported at $p_{T_3}$ will be delivered to all internal ports in composite component $\Delta$ that are connected to port $p_{A_1}$, namely port $p_Y$ of component $Y$ and port $p_Z$ of component $Z$.

### C.3.2 Communication Buses

Communication buses are logical buses to which component ports are linked. The primitives used to establish and to destroy links between a port and a communication bus are the same called to link and to unlink two ports. When a component writes a message to an output port that is linked to a communication bus, this message is delivered to all input ports linked to the same bus. The same procedure applies to request and reply ports. Communication buses are particularly useful when the address of the recipient is unknown.

Communication buses are associations of components which provide access to
"services", rather than to ports. Communication buses can be used to increase the availability of components *transparently* by linking more than one functionally equivalent component to the same bus. Contrary to members of composite components, modules linked to a communication bus are not *part of* it. They can be linked directly to other ports and to other communication buses simultaneously. Elements linked to communication buses can be primitive or composite components or other communication buses.

Figure C.4 illustrates a communication bus. Components A and B are clients linked to communication bus $\Omega$. Requests issued by them are delivered to servers $S_1$ and $S_2$ that are also linked to $\Omega$. However, replies are delivered only to the sender of the request. If both servers are capable of fulfilling the request, both will reply. Duplicate replies are filtered by the kernel module at the sender machine. Redundant replies increase the network traffic, but to try to prevent them would require cancellation messages that could be equally expensive.

### C.4 Primitives

POPE primitives are almost entirely implemented by the kernel. In synchronous communication primitives, the blocking part is implemented at the library level,
which is linked to the component code. Kernel modules are passive elements that are triggered either by primitive calls or by messages sent by other kernel modules.

Kernel primitives are divided into configuration and communication primitives. All primitives return a numeric code which informs the component the status of the operation. Machine, component, and port names are alphanumeric strings of characters. Time-out values are unsigned integers. The largest possible value is used to represent an infinite time-out.

### C.4.1 Configuration Primitives

Configuration primitives are responsible for creating and terminating components, creating, deleting, linking, and unlinking ports. The definitions and syntax of these primitives are as follows:

- **create component**: This primitive registers the component at the kernel. Primitive components are implemented as processes which are actually created by the native operating system.

  **Syntax**: `create_component( machine_name, component_name, component_type )`

  where `component_type` is either a *primitive component*, a *composite component*, or a *communication bus*.

  **Obs.**: In case of composite components and communication buses, the content of the `machine_name` parameter should be a null pointer, because it is not used by the primitive.

- **terminate component**: This primitive removes the register of the component from the kernel. Processes are actually terminated by the native operating system. A component may be *conditionally* or *unconditionally* terminated. In the former, the component is terminated only if none of
its output ports are linked, and if there is no message at input ports waiting to be read. Unconditional termination undoes all existing links and all pending messages are lost.

**syntax:** `terminate_component( machine_name, component_name, mode )`

*where* `mode` is either *conditional* or *unconditional*.

**obs.:** In case of composite components and communication buses, the content of the `machine_name` parameter should be a null pointer, because it is not used by the primitive.

- **create port:** This primitive creates either a primitive component port or a composite component port.

**syntax:** `create_port( machine_name, component_name, port_name, port_type )`

*where* `port_type` is either *output*, or *input*, or *request*, or *reply*.

**obs.:** If the port belongs to a composite component, the content of the parameter `machine_name` should be a null pointer, because it is not used by the primitive.

- **delete port:** This primitive destroys either a primitive component port or a composite component port.

**syntax:** `delete_port( machine_name, component_name, port_name )`

**obs.:** If the port belongs to a composite component, the content of the parameter `machine_name` should be a null pointer, because it is not used by the primitive.

- **link port:** This primitive links a pair of ports of compatible types, i.e., output ports to input ports, and request ports to reply ports. Each of the ports can belong either to primitive component ports or to composite components. The ports can be enumerated in any order. In addition, the
primitive is used to link a port to a communication bus, or to connect two communication buses.

**syntax**: `link_port(machine_name_1, component_name_1, port_name_1, machine_name_2, component_name_2, port_name_2)`

**obs.**: For ports belonging to composite components, the content of the respective `machine_name_n` parameter should be a null pointer, because it is not used by the primitive. For communication buses, the name of the bus is indicated in the `port_name_n` parameter and the contents of the respective `machine_name_n` and `component_name_n` parameters should be null pointers, because they are not used by the primitive.

- **unlink port**: This primitive destroys the link between two ports. Each of the ports can belong either to primitive component ports or to composite components. The ports can be enumerated in any order. In addition, the primitive is used to unlk a port from a communication bus, or to disconnect two communication buses.

**syntax**: `unlink_port(machine_name_1, component_name_1, port_name_1, machine_name_2, component_name_2, port_name_2)`

**obs.**: For ports belonging to composite components, the content of the respective `machine_name_n` parameter should be a null pointer, because it is not used by the primitive. For communication buses, the name of the bus is indicated in the `port_name_n` parameter and the contents of the respective `machine_name_n` and `component_name_n` parameters should be null pointers, because they are not used by the primitive.

- **join composite component**: This primitive establishes the membership of a component (which may be simple or composite) to a composite component. The lower level component cannot be already a member of another
component, nor can it join another composite component whilst its membership lasts.

**syntax**: `join_component(machine_name, member_component_name, composite_component_name)`

**obs.**: If the member component is a composite component, the content of the `machine_name` parameter should be a null pointer, because it is not used by the primitive.

- **leave composite component**: This primitive cancels the membership of a component (which may be simple or composite) to a composite component. The member component can, then, join another composite component.

  **syntax**: `leave_component(machine_name, member_component_name, composite_component_name)`

  **obs.**: If the member component is a composite component, the content of the `machine_name` parameter should be a null pointer, because it is not used by the primitive.

### C.4.2 Communication Primitives

Communication primitives deal with fragmented messages at the presentation layer. Each fragment consists of a string of bytes. Different fragments do not need to be located in adjacent memory addresses. Thus, fragmented messages are normally used when the data transmitted are not in contiguous memory areas, e.g. multiple nodes of a linked list. Fragmentation at the presentation layer, however, does not imply in fragmentation at the lower protocol layers. If the total length of the presentation layer fragments plus the header is smaller than the maximum size of the network layer fragment, the whole message occupies only one fragment of the network layer. Message addresses are passed to the communication primitives by
means of a descriptor which consists of an array of tuples containing the size of each fragment and the address of the respective fragment buffer. Their definitions and syntax are enumerated below.

- **send message**: This primitive transmits messages from the specified output port to all ports linked to it. This is a non-blocking primitive.

  **syntax**: `send_message( port_name, number_of_fragments, message_descriptor )`

- **receive message**: This primitive reads messages from the specified port, which has to be either an input port or a reply port. This primitive blocks the caller for up to the length of time indicated in the `time_out` parameter. A time-out equal to zero indicates a non-blocking receive. The `number_of_fragments` parameter shows the maximum number of fragments that the caller is willing to receive. It is passed by reference, so that it can return the actual number of fragments received. The `message_descriptor` parameter contains the address of the descriptor that points at the addresses of the fragment buffers and their respective maximum expected sizes. Analogously to the `number_of_fragments` parameter, the maximum expected size of each fragment (not visible in the primitive call) is passed by reference. On the completion of the primitive, each of them will contain the actual size of the fragment that they refer to.

  **syntax**: `receive_message( port_name, &number_of_fragments, message_descriptor, time_out )`

- **request a service**: This primitive sends a message from the specified port, which must be a request port, to all ports linked to it. The caller is, then, blocked, for up to the length of time indicated in the `time_out` parameter, waiting for a reply from any of the recipients. The caller is
unblocked when the first reply arrives. A time-out equal to zero indicates a non-blocking receive, implying that the reply will be read further ahead in the computation. The number_of_reply_fragments parameter shows the maximum number of fragments that the caller is willing to receive in the reply message. It is passed by reference so that it can return the actual number of fragments received. The request_descriptor and reply_descriptor parameters have the same description as the message_descriptor in the receive_message primitive. The indication that the caller does not expect a reply is done by a void reply_descriptor parameter. A void reply_descriptor parameter is used in cases when, in addition to request messages, the caller needs to send notification messages via the same request port.

```
syntax: request_message( port_name, 
    number_of_request_fragments, 
    request_descriptor, 
    &number_of_reply_fragments, 
    reply_descriptor, 
    time_out )
```

• reply to a request: This primitive transmits from the specified port, which has to be a reply port, the reply to a request received at the same port to the component which generated the request. This is a non-blocking primitive.

```
syntax: reply_message( port_name, number_of_fragments, 
    message_descriptor )
```

• receive at multiple ports: These primitives allow for the reception of notification and reply messages at multiple ports. A multiple reception block starts with a select call, which informs the kernel about the minimum number of messages that must have arrived before the program leaves the multiple reception block. The select primitive has a second parameter which contains the address of an array of integers where the
return code of each of the primitives inside the multiple reception block will be written. The multiple reception block ends with an end_select primitive, which specifies the deadline for the reception of the first n messages (where n is the first parameter of the select primitive). Primitives receive_message and request_message are placed after the select primitive and before the end_select call. When used within a select-end_select block, the time-out clauses of communication primitives are ignored by the kernel. Statements and loop commands can be used inside a select-end_select block.

**syntax:**
```plaintext
select( minimum_number_of_messages, &return_buffer )
end_select( time-out )
```

- **get a reply:** This primitive reads replies from the specified request port. It is generally preceded by a request_message primitive with a time-out equal to zero or within a select-end_select block. A process may leave a select-end_select block before the replies to all requests issued within the block have arrived. This primitive allows for these replies to be read further ahead in the program. Another reason for separating the reception of the reply from the transmission of the request is found in clients dealing with slow servers. In this case, the client may want to execute other operation whilst waiting for the reply to arrive. Parameters number_of_fragments, message_descriptor, and time_out are equivalent to those in primitive receive_message.

**syntax:**
```plaintext
get_reply_message( port_name, &number_of_fragments, message_descriptor, time_out )
```

- **read the remaining part of a message:** This primitive reads the remaining part of a message sent to the specified port. The port must be an input, request, or reply port. If the number of fragments of an incoming message is greater than the one expected by the caller, only the expected
number of fragments is copied into the caller buffers. However, the total number is returned to the caller, enabling it to allocate more buffer space in order to incorporate the remaining part of the message. This is a non-blocking primitive.

**syntax:** more_message( port_name, &number_of_fragments,
message_descriptor )

- delete the remaining part of a message: This primitive deletes from the specified input, request, or reply port the remaining part of a message sent to that port. It is normally used when the caller decides that the remaining fragments of a message, whose first part it has already incorporated, do not need to be read. However, it can also be called to delete a whole message which has not been read yet.

**syntax:** flush_message( port_name )

### C.5 Examples

An example of a configuration where two clients access a server is presented in this section. The clients send an error message to an alarm component, if the server does not respond to their request within the established deadline. This example illustrates the use of configuration and communication primitives (section C.4) with the complete C code of each of the modules.

Figure C.5 depicts a distributed application consisting of two identical clients (Client1 and Client2), a server component (Server), and an alarm module (Alarm). Client1 and Alarm execute at machine ALPHA, whereas Client2 and Server are located at node BETA. The system is assembled by component Conf at machine ALPHA. There is a copy of the kernel module in each of the machines, despite only user components have been shown in the figure.
Figure C.5: Diagram of a Client-Server-Alarm Configuration

The configuration process could have been instantiated in any of the nodes of the distributed system. A copy of its C code is listed in figure C.6. Conf registers at the kernel before calling other primitives and cancels its registration when it terminates. Whilst all components and ports enumerated in a link_port call have not been created, the primitive returns a value which is different from SUCCESS. File kern_def.h, included in the beginning of the code, contains the definitions of several constants and structures that are used by the kernel internally and to communicate with user components. Components generally include this file.

Server is a component that converts lower case letters in a message into their equivalent upper case representation. Characters out of the range a–z are not altered. Server C code is listed in figure C.7. Server initiates by registering at the kernel and creating reply port read. Port read is used to receive requests and to send replies. No terminate_component call is made at the end of the code, because the service provided by this component is located within an infinite loop. Server expects, by default, messages of up to ten fragments, which are stored in the bi-
Appendix C. POPE: a Port-Oriented Programming Environment

```c
#include "kern_def.h"

void raain()
{
    create_component( "ALPHA", "Configure", SIMPLE );
    while ( link_port( "ALPHA", "Client1", "write", "BETA", "Server", "read" ) != SUCCESS );
    while ( link_port( "BETA", "Client2", "write", "BETA", "Server", "read" ) != SUCCESS );
    while ( link_port( "ALPHA", "Client1", "alarm", "ALPHA", "Alarm", "local" ) != SUCCESS );
    while ( link_port( "BETA", "Client2", "alarm", "ALPHA", "Alarm", "remote" ) != SUCCESS );
    printf( \nPorts Linked." );
    terminate_component( "ALPHA", "Configure", ANYWAY );
} /* main */
```

Figure C.6: Example of a Configuration Component

dimensional byte array `frag`. Messages are read at the `receive_message` statement. The addresses of the fragments and their respective sizes are stored in `rec_buffer`, which is an array of `struct com_data`. This structure, defined in file `kern_def.h`, consists of two elements: a byte array pointer, which is used to store the address of the fragment buffer, and an integer, which contains the fragment size. If the actual number of fragments in the message is greater than ten, `Server` reads the remaining fragments by calling the `more_message` primitive. The first ten fragments are written at `frag`, but before `Server` reads the remaining fragments, it allocates extra buffer space for them. After the conversion is done, the upper case message is written at `read` by the primitive `reply_message` and the distributed kernel delivers it to the port at which the request originated. If extra buffer space has been allocated, `Server` frees it before returning to the beginning of the loop.

The C code of client component `Client1` is shown in figure C.8. Had strings "ALPHA" and "Client1" been parameterised, `Client1` and `Client2` could have been instances of the same client program at different machines. As in any application component, `Client1` registers at the kernel and creates the ports at which it will
#include "kern_def.h"

void main( )
{
  struct com_data  rec_buffer[10],
  *p_rec_buffer;
  char     frag[10][80+1],
  *ext_frag;
  unsigned  n_rec_msg;
  unsigned  i, j;

  create_component( "BETA", "Server", SIMPLE );
  create_port( "BETA", "Server", "read", RINPUT );
  for (  i = 0; i < 10; i++ )
    rec_buffer[i].frag_addr = &frag[i][0];
  for (;; ) /* ever */
  {
    for (  i = 0; i < 10; i++ )
      rec_buffer[i].frag_size = 80+1;
    n_rec_msg = 10;
    receive_message( "read", &n_rec_msg, rec_buffer, INFINITE );
    if ( n_rec_msg <= 10 )
      p_rec_buffer = rec_buffer;
    else
      {
        p_rec_buffer = malloc( n_rec_msg * sizeof(struct com_data) );
        ext_frag = malloc( (n_rec_msg-10) * (80+1) );
        for (  i = 0; i < n_rec_msg; i++ )
          {
            p_rec_buffer[i].frag_addr = rec_buffer[i].frag_addr;
            p_rec_buffer[i].frag_size = rec_buffer[i].frag_size;
          } /* for: i */
        n_rec_msg -= 10;
        more_message( "read", &n_rec_msg, &p_rec_buffer[10] );
        n_rec_msg += 10;
      } /* else */

    for (  i = 0; i < n_rec_msg; i++ )
      for (  j  = 0; j  < p_rec_buffer[i].frag_size; j++ )
        if ( (p_rec_buffer[i].frag_addr[j] >= 'a') &&
            (p_rec_buffer[i].frag_addr[j] <= 'z') )
          p_rec_buffer[i].frag_addr[j] -= ( 'a' - 'A' );
    reply_message( "read", n_rec_msg, p_rec_buffer );
    if ( n_rec_msg > 10 )
      {
        free( ext_frag );
        free( p_rec_buffer );
      } /* if: n_rec_msg */
  } /* for ever */
} /* main */

Figure C.7: Example of a Server Component
communicate as part of its initiation procedure. $Client_1$ reads strings of characters from the keyboard into bi-dimensional byte array $frag$. It, then, sends these strings to the modules linked to port $write$ ($Server$, in this application) by making a $request\_message$ call. As a result, it is blocked for up to thirty seconds, waiting for the reply. When the first reply arrives, $Client_1$ is unblocked and the reply is printed. In this example, only one reply will ever be produced for each request, because there is only one server linked to $write$. However, should more servers have been linked to that port, there would be up to as many replies as the number of operational servers. In the example of figure C.8, the number of reply fragments matches that of the request message. However, a different server could issue a reply with a different number of fragments, in which case $Client_1$ would need to call the $more\_message$ primitive, if the number of fragments of the reply message has exceeded that of the request. $Client_1$ creates another port, namely $alarm$, to communicate with alarm modules. An error message is sent via this port by a $send\_message$ call, if the client server component does not reply within the pre-established deadline of thirty seconds.

The final component of the configuration shown in figure C.5 is $Alarm$. $Alarm$ is a $consumer$ component which reads messages from two different input ports, namely $local$ and $remote$, and prints them on the screen. By inspecting $Alarm$ C code (figure C.9), one can see that $Alarm$ blocks at the $end\_select$ call until, at least, one message arrives either at port $local$ or at port $remote$. The use of two distinct ports enables the component to establish the type of the message. From the algorithm, one can infer that $Alarm$ deals with two classes of error messages: one received from local components, and another coming from other sites. Because the process will block only at the $end\_select$ call, the kernel disregards the time-out clause of the $receive\_message$ primitives. In this example, this value was replaced by the constant $VOID$ that is defined in file kern_def.h.
#include "kern_def.h"

void main( )
{
    struct com_data request_buffer[50],
                      alarm_buffer;
    char        frag[50][80+1];
    unsigned   n_frags;
    unsigned i;
    PRIMITIVE_RET_CODE ret;

    create_component( "ALPHA", "Client1", SIMPLE );
    create_port( "ALPHA", "Client1", "write", ROUTPORT );
    create_port( "ALPHA", "Client1", "alarm", OUTPORT );
    for ( i = 0; i < 50; i++ )
        request_buffer[i].frag_addr = &frag[i][0];
    alarm_buffer.frag_addr = "SERVER does not respond";
    alarm_buffer.frag_size = strlen( "SERVER does not respond" ) + 1;
    for ( EVER )
    {
        printf( "Enter the number of request fragments: ");
        scanf( "/*,.d", &n_frags );
        for ( i = 0; i < n_frags; i++ )
        {
            printf( "Enter fragment #*/.d: ", i );
            scanf( "/*s", &frag[i][0] );
            request_buffer[i].frag_size = strlen( &frag[i][0] ) + 1;
        } /* for */
        ret = request_message( "write", n_frags, request_buffer,
                           &n_frags, request_buffer, 30 );
        if ( ret == TIME_OUT )
            send_message( "alarm", 1, &alarm_buffer );
        else
            for ( i = 0; i < n_frags; i++ )
                printf( "Converted fragment #*/d: %s", i, &frag[i][0] );
    } /* for ever */
} /* main */

Figure C.8: Example of a Client Component
 Appendix C. POPE: a Port-Oriented Programming Environment

#include "kern_def.h"

void main()
{
  struct com_data alarm_buffer[2];
  char alarm_msg[2][80+1];
  unsigned n_alarm_frags[2];
  PRIMITIVE_RET_CODE ret[2];

  create_component("ALPHA", "Alarm", SIMPLE);
  create_port("ALPHA", "Alarm", "local", INPORT);
  create_port("ALPHA", "Alarm", "remote", INPORT);
  alarm_buffer[0].frag_addr = &alarm_msg[0][0];
  alarm_buffer[1].frag_addr = &alarm_msg[1][0];
  for ( ; ; ) /* ever */
  {
    alarm_buffer[0].frag_size =
    alarm_buffer[1].frag_size = 80 + 1;
    n_alarm_frags[0] =
    n_alarm_frags[1] = 1;
    select( 1, ret );
    receive_message("local", &n_alarm_frags[0], &alarm_buffer[0], VOID);
    receive_message("remote", &n_alarm_frags[1], &alarm_buffer[1], VOID);
    end_select( INFINITE );
    if ( ret[0] != NO_QUEUED_MESSAGE )
    {
      printf( "\n\n\nThe machine where the server is located may have crashed!" );
    } /* if: ret[0] */
    if ( ret[1] != NO_QUEUED_MESSAGE )
    {
      printf( "\n\n\nThe machine where the server is located is operational." );
    } /* if: ret[1] */
  } /* for ever */
} /* main */

Figure C.9: Example of a Consumer Component
C.6 Summary

POPE is a port-oriented programming environment based on the programming-in-the-small versus programming-in-the-large approach. It caters for the development of a framework for designing, implementing, and testing software components that are capable of tolerating residual design faults in the code. However, POPE is not restricted to software fault-tolerant applications.

In POPE, processes are primitive components that encapsulate data. Hence, data can only be obtained or modified by communicating with the component that contains them. Processes communicate by synchronous or asynchronous message exchange. Messages are written to and read from local named ports. Process configuration and communication is implemented by a distributed kernel, which is transparent to the user.

Components can form clusters in order to produce a higher level abstraction. Clusters can either be composite components or communication buses. Members of composite components are encapsulated and, therefore, cannot be accessed directly by other elements in the system. Conversely, there is no membership relationship between communication buses and the components linked to them. Composite components are made of lower level modules that are bound together, whereas communication buses are logical connection buses that may exist without any module linked to them.
Appendix D

Principles of Air Traffic Control

The significant number of aircraft which share air spaces in some areas of the globe demands strong flight co-ordination. This co-ordination is done by air traffic control units, which can be divided in three types [Int90]: area control centre, approach control office, and aerodrome control tower. The area control centre is established to provide air traffic control service to flights in control areas under its jurisdiction. The approach control office provides control service to flights arriving at and departing from one or more aerodromes. The aerodrome control tower co-ordinates flights on the manoeuvring area of an aerodrome or flying in its vicinity, either intending to land at the aerodrome or having just taken off from it. More than one of these activities may be performed by the same unit.

The air space under the jurisdiction of an area control centre is divided into control sectors. An air traffic controller is responsible for each control sector. Flights entering a control sector can come either from an aerodrome or from an adjacent control sector. In either case, the controller is informed that the flight will enter his/her sector some time in advance. The information about the flights under the responsibility of an air traffic controller or due to enter his/her control sector in the near future is recorded on strips of paper, or simply strips in the technical jargon. Analogously, the control of a flight transferred to an approach control office if the flight terminates at a local aerodrome, or to another control sector, otherwise.
D.1 Main Air Traffic Control Activities

An air traffic control (ATC) unit must be capable of knowing the intention and position of an aircraft in order to accomplish its goals, which are [Int90]:

- "to prevent collisions between aircraft";
- "to prevent collisions between aircraft on the manoeuvring area and obstructions on the area";
- "to expedite and maintain an orderly flow of air traffic";
- "to provide advice and information useful for the safe and efficient conduct of flights";
- "to notify appropriate organisations regarding aircraft in need of search and rescue aid, and assist such organisation as required".

The intention of a controlled flight is made known to the ATC unit in the form of a flight plan, which is delivered to the ATC unit by the pilot or someone else on his/her behalf prior to the departure of the aircraft, or before it enters the control area from an adjacent control area. The flight plan contains the details of the flight, e.g., cruise velocity, flight level, aerodromes of departure and destination, planned departure time and expected arrival time, intermediate points at which the characteristics of the flight, such as cruise velocity and flight level, will be altered, etc. Once the pilot’s intention is known, the ATC unit is able to plan the flight schedule. The ATC unit, however, has to ensure that either the original plan is followed or that it will be properly amended and that no conflict will arise from this update. This can be achieved by radio communication with the pilot at pre-established points of the flight (report points) and/or by tracking the position of the aircraft in real-time via surveillance radars.
D.1.1 Analysis of Flight Plans

The flight plan is the means by which the ATC unit is informed of the intention of the pilot during the flight. A flight plan can be received by the ATC unit on paper, by telephone, or by radio. It is then manually input, usually by an operator (instead of the controller) in order to reduce the workload on the latter. Alternatively, if the flight plan has been filed elsewhere, i.e., if it is a filed flight plan (FPL), it can be transmitted to the ATC unit via the aeronautical fixed communication network (AFNT). Scheduled flights have repetitive flight plans (RPL). In an automated air traffic control system, repetitive flight plans can be stored locally and retrieved at the appropriate time.

Amongst the fields of a flight plan [Int86], the most important ones, as far as ordinary air traffic control is concerned, are: aircraft or flight identification, type of aircraft, wake turbulence class, aerodrome of departure, planned departure time, original cruising speed and level, flight route, aerodrome of destination, and expected time of arrival. Figure D.1 depicts a flight plan form.

The wake of turbulence class serves to determine the minimum horizontal separation that the aircraft must keep from the one ahead of it when both are approaching the runway for landing or when they are taking off. This information is needed, for instance, when arranging the order in which arriving aircraft are to line up to land.

The planned departure time and expected arrival time are used for scheduling the departures and arrivals at the respective aerodromes. However, it is not uncommon to see flights that do not comply with the timetable. The time of departure of a flight can be altered, usually delayed, for a number of reasons, such as congestion in the aerodrome area and technical problems with the aircraft. The time of arrival is estimated from the cruise speed planned for each section of the flight without taking into account the wind velocity during the flight. Since this assumption does not hold throughout the flight, the aircraft may arrive earlier or later than expected.
Figure D.1: Example of a Flight Plan Form
The flight route is specified in terms of points at which the pilot is supposed to establish radio contact with the air traffic controller. Air routes are predefined imaginary lines in the sky along which aircraft fly. Report points which belong to air routes are referred to by names. Other report points are identified by their latitude and longitude. When an aircraft follows a section of an air route, the report points along that section do not need to be enumerated in the route field of the flight plan, unless the cruise level and/or speed of the aircraft are supposed to change there. The air route name is used instead.

The extraction of the route consists of identifying the sequence of report points that will be flown by the aircraft. When the route has been extracted, the approximate time at which radio contact will be established between the pilot and the controller is calculated for each report point. The time taken by an aircraft to fly from one report point to the next is calculated according to its current cruise speed and corrected by adding the vectorial component of the wind velocity along the aircraft trajectory. The wind data is received periodically from the meteorological office. Should the difference between the estimated and actual times of report at a particular point be greater than a pre-established value, the estimated report times at the next points are re-calculated.

D.1.2 Analysis of Radar Data

The actual position of an aircraft can be determined through radio contact with the pilot. However, in areas of dense concentration of air traffic, a visual representation of the location of the aircraft may be crucial. As mentioned earlier in this section, the air traffic controller is provided with a visual representation of the actual position of the aircraft in his/her control sector on a radar screen. This enables the controller to ensure that the various flights follow the route described in their flight plans and, more important, to avoid collisions between aircraft.

Document 4444-RAC/501 (Rules of the Air and Air Traffic Services) [Int78] of
the International Civil Aviation Organisation (ICAO) establishes separation minima
between aircraft in order to prevent collision during flight. Briefly, aircraft must be
separated vertically by at least 1,000 feet, if they are below 29,000 feet and separated
vertically by at least 2,000 feet, if they are above this level. Aircraft flying on the
same level must keep a distance of at least 5 NM (nautical miles\(^1\)) in any direction
from one another. Data received from the primary and secondary surveillance radars
are analysed in order to monitor the distances amongst the aircraft in the control
sector. Should violations of the separation minima be detected, the controller must
instruct the pilots to change course in order to avoid a collision.

**D.2 Automated Air Traffic Control**

In areas of intense air traffic, the load and stress on the air traffic controller can
be considerable. In order to keep the amount of air traffic co-ordinated by each
controller at acceptable levels, the air space can be divided into more control sectors
of smaller sizes. However, the smaller the control sector, the more frequent the
flight transfers. Thus, the benefit of diminishing the number of flights under the
responsibility of a controller by reducing the size of the control sector is limited
by the increase in the number of flight transfers. An alternative and more efficient
means of easing the stress on the controller is to introduce automation into air
traffic control [Int]. In modern automated air traffic control systems (ATCS), the
computing system aids air traffic controllers on a number of tasks, but the final word
is theirs.

In ATC units, primary radar tracks are displayed on a screen which is located
in front of the controller. Some of these tracks, however, are false echoes caused by
other obstacles, such as flocks of birds and clouds. In order to aid the controller,

\(^1\)One nautical mile is, by definition, equal to one minute of longitude measured along the
equator, which is equal to 6,082.66 feet (approximately 1,854 km).
automated air traffic control systems show the aircraft or flight identification along- side its primary radar track. This label can be entered manually by the controller. Alternatively, the identification of aircraft equipped with transponders (electronic devices that record the flight number assigned to the aircraft) can be obtained automatically from secondary surveillance radars (SSR), which read the signals emitted by the transponders.

In conventional air traffic control, the paper strips where data about controlled flights are recorded are filled out manually by the air traffic controller. They are placed in mobile sockets so that they can be ordered and easily rearranged according to changes in the flight. Paper strips could be replaced by electronic versions which would be more effectively filled out, updated, and sorted by computer systems. However, even in ATCSs in which electronic strips have been implemented, the paper version has been maintained due to the resistance of many controllers to work with a purely electronic version. At present, in automated air traffic control systems, the paper strips are filled out by the computer system, but the air traffic controller continues to update and arrange them manually.
Glossary

acceptance test: piece of code, procedure, or module responsible for the assessment of the results of the execution of the alternatives in a recovery block.

action: the doing of something. Changes in values of variables and registers are known as internal actions. Actions which cause modifications in external devices or which are noted by the issue of messages are known as external actions.

active replication: fault-tolerance technique which caters for the replacement of a module which has failed by an equivalent (usually identical) one which has been executing simultaneously to the original one, subject to the same inputs.

adjudicator: see acceptance test.

alternative: diversely designed solution of a problem, usually used with other alternatives in recovery block techniques.

assessment algorithm: algorithm which assesses the outputs of the modules in software fault-tolerance mechanisms based on design diversity in order to produce a consensus output. The algorithm may also indicate those results which agree with the consensus output and those which do not.

ATC: air traffic control.

ATCS: air traffic control system.
**Glossary.**

**atomic action**: software fault-tolerance technique in which changes in the state of the module made within a well delimited piece of code are made permanent only when the whole piece of code is executed correctly.

**atomic broadcast/multicast protocol**: class of broadcast/multicast protocol in which every message broadcast/multicast by a correct sender is delivered to all correct receivers or to none, within some known time bound, and in the same order.

**availability**: the degree of confidence on the correct behaviour of the system at a particular time. In mathematical terms, it has the form of the function $A(t)$ which describes the probability that the system will be operating according to the specification at time $t$, regardless of whether it has failed previously or not.

**backward error recovery**: class of fault-tolerance technique which takes the system from an erroneous state back into a previous, correct one.

**catastrophic failure**: failure which produces unpredictable results to the rest of the system.

**checkpoint**: place in the code at which the current state of the module is saved.

**cold stand-by replication**: fault-tolerance technique which caters for the replacement of a module which has failed by an equivalent (usually identical) one, without preserving the state of the original one.

**collation**: see voting.

**collator**: see voter.

**compensation**: software fault-tolerance technique where supplementary computation is executed in order to compensate an action which has been incorrectly executed previously.
component: module or set of modules which serves as part of a system. A primitive component is a component which cannot be divided into subcomponents that still keep the same type of interface. Conversely, a composite component is a system made of more than one subcomponent which is used as a building block for a higher level system.

community error recovery: recovery technique used in software fault-tolerance mechanisms based on design diversity whereby the outcome of the assessment algorithm is used to correct the value of the modules whose outputs were rejected.

contamination: transmission of incorrect results produced by a faulty module to others in the system.

conversation: logical area formed by pieces of code belonging to communicating modules established such that all communication amongst the modules occurs only when all of them have entered their respective piece of code. In addition, all modules must leave the conversation at the same time. Also the name of the software fault-tolerance technique based on such structure, designed to avoid the domino effect.

critical point of failure: component or part of it in which a failure leads the whole system to collapse.

design diversity: principle of designing equivalent solutions of the same problem, each according to a dissimilar strategy.

domino effect: anomalous effect of rolling back concurrent communicating modules further than their last checkpoints.

environment: physical or logical system which stimulates another system with inputs, causing a state transition on that system. In some cases, the new state can be the same as the original one. Besides the state transition, the system stimulated by the environment can produce an output to the environment.
Glossary.

error: manifestation of defects in the system.

time error: class of fault-tolerance technique which prevents the propagation of an erroneous output by choosing the outputs of an equivalent module.

event: occurrence which causes a state transition in a system.

event handler: especially designed piece of code which will be called when an abnormal situation in the computation occurs.

exception: abnormal situation in the computation.

exception handler: especially designed piece of code which will be called when an abnormal situation in the computation occurs.

exception handling: software fault-tolerance technique in which a especially designed piece of code will be called when an abnormal situation in the computation occurs.

fail-safe mode: failure mode in which, despite any possible degradation in performance, the system continues to work according to the specifications.

failure: deviation of the system from its specification.

time fault: defect in the system. Faults which cause the system to fail to perform an action when one is expected are known as omission faults. Faults which cause the system to execute an action sooner or later than acceptable are known as timing faults. Faults which cause the system to carry out an erroneous action are known as byzantine faults.

fault-tolerance: to provide, by redundancy, service complying with the specification in spite of faults having occurred or occurring ([Lap85]).

forward error recovery: class of fault-tolerance technique which takes the system from an erroneous state into a new, correct one.

history-dependent module: module that depends on its previous states to accomplish with its specification. Module that retains state. Idempotent module.
**Glossary.**

history-independent module: module which does not depend on its previous states to accomplish with its specification. Stateless module.

hot stand-by replication: fault-tolerance technique which caters for the replacement of a module which has failed by an equivalent (usually identical) one, preserving the state of the original one.

interaction: communication between two or more modules triggered by a message sent by one of them and terminated either when the message is received by all addressees (*notification interaction*) or when a reply issued by one of the addressees is received by the module which initiated the interaction (*request-reply interaction*).

multi-version programming: see *n-version programming*.

**MTBF:** (for Mean Time Between Failures): the average time interval between two consecutive failures in the system.

**MTTF:** (for Mean Time To Fail): the average time interval in which the system complies continuously with the specification.

**MTTR:** (for Mean Time To Repair): the average time interval when the system is out of work due to a failure.

native operating system: any general purpose operating system to which extra functions are added.

**n-version programming:** software fault-tolerance technique based on the concurrent execution of diversely designed versions which perform equivalent functions and the collation of their internal states at checkpoints.

passive replication: fault-tolerance technique which caters for the replacement of a module which has failed by an equivalent (usually identical) one which starts to execute from the very beginning (cold stand-by replication) or from the last checkpoint (hot stand-by replication).
**Glossary.**

**port:** local-named entities in modules which communicate by message exchange where messages are written to and read from. **output port:** port where messages that do not require any response are written to. **input port:** port where messages that do not require any response are read from. **request port:** port where messages that require a response are read from and from which the reply is read. **reply port:** port where messages that require a response are read from and to which the reply is written.

**real time:** physical time.

**real-time:** denoting a physical time constraint.

**recovery block:** programming structure consisting of diversely designed alternatives which perform equivalent functions and an acceptance test. Each of the diversely designed alternatives is executed at a time and the results are assessed by the acceptance test. If the results are rejected, the system rolls back to the state at the beginning of the recovery block and the next alternative in line is executed. This procedure is repeated either until an alternative is successful or until no more alternatives are left. In the latter case, an error indication is returned. Also the name of the software fault-tolerance technique based on this form of structure.

**reliability:** the degree of confidence on the continuous correct behaviour of the system. In mathematical terms, it is represented as the function $R(t)$ that describes the probability that the system will have not failed by time $t$.

**reliable output or result:** consensus output or result achieved by an assessment algorithm.

**replica:** module which accomplishes with the same specification of another. A replica can be an exact copy or the original module or vary in design.

**replication:** fault-tolerance technique whereby replicas of a module are made available for a possible replacement of the original one, if it fails.
rollback: procedure of restoring the process to a previous state.

software module: process or set of concurrent processes which perform together a particular function. A software module does not share data or state with other modules.

state: set of observable values that influence the behaviour of the system. The external state of a system is known as its external behaviour (i.e., the way that it is seen by the environment). The internal state of a system is represented by the value of its variables and registers.

stateless module: see history-independent module.

subsystem: the same as component, although usually denoting one made of multiple modules.

system: set of components which interact under the control of a design ([AL81]).

transaction: technique used in database systems to ensure consistency of the database whereby changes in the data made within a well delimited piece of code are made permanent only when the whole piece of code is executed correctly.

version: diversely designed solution of a problem, usually used with other versions in error masking software fault-tolerance techniques.

voter: piece of code, procedure, or module responsible for the voting.

voting: process by which equivalent values in diversely designed versions are compared in order to produce a unique value.
Bibliography


International Civil Aviation Organization (ICAO). *Air Traffic Services Planning Manual, Part II: Methods of application employed by Air Traffic Services, Section 3, Chapter 3: Application of automation in air traffic services.*


