Fault-Tolerant Software: Dependability/Performance Trade-Offs, Concurrency and System Support

Jie Xu

Ph.D. Thesis

Department of Computing Science
University of Newcastle upon Tyne

September 1999

BLANK PAGE
IN
ORIGINAL

Abstract

As the use of computer systems becomes more and more widespread in applications that demand high levels of dependability, these applications themselves are growing in complexity in a rapid rate, especially in the areas that require concurrent and distributed computing. Such complex systems are very prone to faults and errors. No matter how rigorously fault avoidance and fault removal techniques are applied, software design faults often remain in systems when they are delivered to the customers. In fact, residual software faults are becoming the significant underlying cause of system failures and the lack of dependability. There is tremendous need for systematic techniques for building dependable software, including the *fault tolerance* techniques that ensure software-based systems to operate dependably even when potential faults are present. However, although there has been a large amount of research in the area of fault-tolerant software, existing techniques are not yet sufficiently mature as a practical engineering discipline for realistic applications. In particular, they are often inadequate when applied to highly concurrent and distributed software.

This thesis develops new techniques for building fault-tolerant software, addresses the problem of achieving high levels of dependability in concurrent and distributed object systems, and studies system-level support for implementing dependable software. Two schemes are developed – the $t/(n-1)$-VP approach is aimed at increasing software reliability and controlling additional complexity, while the SCOP approach presents an adaptive way of dynamically adjusting software reliability and efficiency aspects. As a more general framework for constructing dependable concurrent and distributed software, the Coordinated Atomic (CA) Action scheme is examined thoroughly. Key properties of CA actions are formalized, conceptual model and mechanisms for handling application level exceptions are devised, and object-based diversity techniques are introduced to cope with potential software faults. These three schemes are evaluated analytically and validated by controlled experiments. System-level support is also addressed with a multi-level system architecture. An architectural pattern for implementing fault-tolerant objects is documented in detail to capture existing solutions and our previous experience. An industrial safety-critical application, the Fault-Tolerant Production Cell, is used as a case study to examine most of the concepts and techniques developed in this research.

Key Words — Concurrent and Distributed Systems, Exception Handling, Industrial Production Cell, Multi-Version Software, Object Systems, Software Architecture, Software Fault Tolerance.

"To my wife Mei,
my daughter Melanie-Xin,
my sister and parents"

Acknowledgements

I am greatly indebted to my supervisor, mentor, and role model, Prof. Brian Randell, for his help, encouragement, criticism, and professionalism throughout every stage of the development of my research work. I would also like to thank my colleagues in the Dependability Group at Newcastle, Robert J. Stroud, Alexander Romanovsky, Avelino F. Zorzo, Ian S. Welsh, Cecilia M.F. Rubira and Zhixue Wu, for their collaborative roles in a stimulating and friendly research environment, and for many constructive discussions and meetings over the years of three ESPRIT research projects. Indeed, I should like to thank all the colleagues within these projects who have helped me directly or indirectly in my research efforts.

I would like to thank the members of the Department of Computing Science at Newcastle. In particular, I am grateful to Shirley Craig for helping me to navigate our libraries, and also to Barry Hodgson for his support and help during these years.

My work presented in this thesis has been supported mainly by three successive ESPRIT research projects on Predictably Dependable Computing Systems (PDCS and PDCS2 under grant numbers 3092 and 6362) and on Design for Validation (DeVa under grant number 20072).

Finally, my special thanks and apologies must go to my dearest family, Mei, Xin-Xin, my sister and parents, who over the years have been neglected during my deep concentration. Without their constant support and encouragement, my entire work would not have been possible.

Paginated blank pages are scanned as found in original thesis

No information is missing

CONTENTS

Abstract i
Acknowledgements v
List of Figures x
List of Tables xiii
List of Programs xiv

1 Introduction 1
1.1 Motivation 1
1.2 Goals of This Research 3
1.3 Thesis Overview 3
1.3.1 Main Results 4
1.3.2 Thesis Structure and Related Publications 5

2 Fault-Tolerant Software 7
2.1 Basic Terminology and Concepts 7
2.1.1 System Model 7
2.1.2 Dynamic Behaviour of Software Systems 9
2.1.3 Software Faults, Errors and Failures 9
2.1.4 Origins of Software Faults 9
2.1.5 Software Fault Tolerance 10
2.1.6 Various Measures 11
2.1.7 Cost, Effectiveness and Efficiency 13
2.1.8 System Support for Fault-Tolerant Software 13
2.2 Fault Tolerance in Sequential Software 14
2.2.1 Techniques for Dealing with Software Faults 14
2.2.2 Basic Schemes: Recovery Blocks and *N*-Version Programming 15
2.2.3 Advanced Schemes 21
2.2.4 Dependability and Cost Effectiveness 26
2.2.5 Analytical Evaluation and Experimental Validation 29
2.2.6 Industrial Applications 33
2.3 Fault Tolerance in Concurrent Software 36
2.3.1 Models of Constructing Concurrent Fault-Tolerant Software 36
2.3.2 Process-Oriented Systems and Conversations 37
2.3.3 Object-Based Systems and Atomic (Trans)actions 42
2.3.4 Necessity of an Integrated Framework 45
2.4 System-Level Support and Environments 49
2.4.1 Typical System Examples 49
2.4.2 Reusable Components Supporting Software Fault Tolerance 51
2.4.3 Reflective System Architecture 53
2.5 Summary 54

3 Advanced Schemes for Designing Fault-Tolerant Software 57
3.1 $t/(n-1)$-Variant Programming 57
3.1.1 Description of the $t/(n-1)$-VP Scheme and an Example 58
3.1.2 General $t/(n-1)$-VP Architecture 61
3.1.3 Comparison with Other Schemes 63
3.2 Self-Configuring Optimal Programming 64
3.2.1 Software Fault Tolerance versus Software Efficiency 64
3.2.2 The SCOP Scheme 67
3.3 Analytic Evaluation of Fault-Tolerant Software 74
3.3.1 Evaluation of $t/(n-1)$-VP 76
3.3.2 Evaluation of SCOP 83
3.4 Empirical Comparison 88
3.4.1 C++ Implementation of $t/(n-1)$-VP and SCOP 89
3.4.2 Timing Results Based on Software-Fault Injection 91
3.5 Summary 94

4 Fault Tolerance in Concurrent Object-Oriented Software 95
4.1 Abstract Model for Concurrent/Distributed Systems 96
4.1.1 Objects, Threads and Actions 96
4.1.2 A Realistic Example: The Arjuna System 97
4.2 Coordinated Atomic Actions Revisited 98
4.2.1 Informal Description and an Example 98
4.2.2 Temporal Logic 101
4.2.3 Elementary Properties of CA Actions 102
4.2.4 Enclosure, Unidirectional Enclosure and Non-Enclosure 104
4.2.5 Exceptions and Error Recovery 107
4.3 Exception Handling in Concurrent/Distributed Systems 108
4.3.1 Conceptual Model 110
4.3.2 Dealing with Environmental Exceptions 112
4.3.3 Dealing with Concurrent Exceptions 118
4.4 Tolerating Software Faults by Design Diversity 127
4.4.1 Object Diversity 127
4.4.2 Adaptive Recovery and Fault Masking: Two Schemes 130
4.4.3 Prototype Implementation and Empirical Study 135
4.5 Summary 139

5 System-Level Support for Implementing Fault-Tolerant Software 141
5.1 Architectural Support 142
5.1.1 Fault-Tolerant Components 143
5.1.2 Supporting the Development of Fault-Tolerant Components 146
5.1.3 Multi-Level Reference Architecture 148

5.2 Architectural Pattern for Implementing Fault-Tolerant Objects 152
5.2.1 Pattern: Generic Software Fault Tolerance (GSFT) 153
5.2.2 GSFT Structure 155
5.2.3 GSFT Implementation 159
5.2.4 Consequences of Using GSFT 164
5.3 Case Study: The Fault-Tolerant Production Cell 167
5.3.1 Description of the Fault-Tolerant Production Cell 168
5.3.2 Assumptions and Failure Analysis 170
5.3.3 Design of a Control Program Using CA Actions 174
5.3.4 Dealing with Software Faults in the Control Program 179
5.3.5 Dealing with Hardware Component Failures in the Cell 183
5.3.6 Implementation of the Control Program 187
5.3.7 Experience and Lessons 192
5.4 Summary 194

6 Conclusions 197
6.1 Major Contributions 197
6.2 Directions for Future Research 198
6.2.1 *N*-Version Design versus One Good Version 198
6.2.2 Real-Time CA Actions 200
6.2.3 Diverse Security Measures 200
6.2.4 Pattern-Oriented Architecture and Systems 201
6.3 In Conclusion 201

Appendixes 203
A Correctness of the $t/(n-1)$-VP Scheme 203
B Proofs of the Correctness of Algorithm 4.1 207
C Communication Complexity of Algorithm 4.1 209

References 211

List of Figures

Figure 2.1 System and components

Figure 2.2 System's dynamic behaviour and erroneous transition T_e

Figure 2.3 Origins of software faults (a recursive view)

Figure 2.4 The dependability tree

Figure 2.5 An example of supporting systems

Figure 2.6 Recovery block syntax and its operation

Figure 2.7 NVP architecture

Figure 2.8 DRB architecture

Figure 2.9 NSCP architecture ($N = 4$)

Figure 2.10 CRB architecture

Figure 2.11 AV architecture

Figure 2.12 The domino effect

Figure 2.13 Nested conversations

Figure 2.14 An FT-Action

Figure 2.15 Three processes in a colloquy of three dialogs

Figure 2.16 Transaction and nested transactions

Figure 2.17 Multi-threaded concurrent transactions in Venari

Figure 2.18 Two-way communication: transactions versus CA actions

Figure 2.19 Control transfer between an application program and the RMP

Figure 2.20 An example of the module definition for RB

Figure 2.21 An example of a single process structured as a set of RP-actions

Figure 2.22 An example of implementing error recovery in C++

Figure 2.23 Invocation trapping in Open C++

Figure 3.1 A $t/(n-1)$-VP architecture with $n = 5$ and $t = 2$

Figure 3.2 Execution model of the $t/(n-1)$-VP scheme

Figure 3.3 Space and time redundancy in fault-tolerant software

Figure 3.4 Execution model of the SCOP scheme

Figure 3.5 Dynamic behaviour of SCOP in a varying environment

Figure 3.6 A modified behaviour model

Figure 3.7 $t/(n-1)$-VP model

Figure 3.8 NVP model

Figure 3.9 NSCP model

Figure 3.10 A simple behaviour model

Figure 3.11 SCOP model

Figure 3.12 Target environment (hardware/software)
Figure 3.13 Class hierarchy of array sorting
Figure 3.14 Impact upon execution times of fault-tolerant software

Figure 4.1 The Arjuna system
Figure 4.2 Possible outcomes of a CA action
Figure 4.3 Example of a CA action
Figure 4.4 Action *c* and the bypass role *i*
Figure 4.5 Exception propagation over nesting levels
Figure 4.6 Dealing with exceptions through the software lifecycle
Figure 4.7 Class-Responsibilities-Collaborators cards
Figure 4.8 Solution structure
Figure 4.9 Collaborations between objects
Figure 4.10 Example of a four-level exception graph
Figure 4.11 Concurrent exception handling and resolution
Figure 4.12 Architecture for a prototype implementation
Figure 4.13 Effect on total execution time
Figure 4.14 Performance-related comparison of our algorithm and the CR algorithm
Figure 4.15 Adaptive recovery scheme
Figure 4.16 Fault masking scheme
Figure 4.17 Two types of CA actions that use diverse threads
Figure 4.18 Layered diagram of the prototype system

Figure 5.1 An idealized fault-tolerant component [Anderson & Lee 1981]
Figure 5.2 Extended interface of a fault-tolerant component
Figure 5.3 A fault-tolerant component with diverse design
Figure 5.4 Reference architecture for fault-tolerant applications
Figure 5.5 Reflective architecture in a distributed environment
Figure 5.6 Structure of the proposed pattern
Figure 5.7 Extended structure of concurrent fault-tolerant software
Figure 5.8 Structure for the use of low-level services
Figure 5.9 Interaction diagram for $t/(n-1)$-VP
Figure 5.10 Interaction diagram for CA actions using FM
Figure 5.11 Control structure for CA actions using AR
Figure 5.12 Recursive combination of adjudication functions
Figure 5.13 The Fault-Tolerant Production Cell (top view)
Figure 5.14 CA actions that control the Fault-Tolerant Production Cell
Figure 5.15 CA action `LoadPress1`
Figure 5.16 `LoadPress1` as container action to tolerate software faults

Figure 5.17 Exception graph for CA action `LoadPress1`
Figure 5.18 Interaction between controllers and CA actions
Figure 5.19 Revised failure injection panel
Figure 6.1 Software reliability versus cost
Figure A.1 Examples of $t/(n-1)$-fault diagnosable systems

List of Tables

Table 2.1 Comparison of RB and NVP

Table 2.2 Extra cost of fault-tolerant software

Table 3.1 Possible syndromes and result selections for the 2/(4)-VP example

Table 3.2 Execution examples of SCOP

Table 3.3 Major implementation characteristics of SCOP, NVP and RB

Table 3.4 Notation for dependability evaluation

Table 3.5 Specific expressions for q_l's and q_U's

Table 3.6 Comparison of resource consumption

Table 3.7 Resource consumption of SCOP and NVP

Table 3.8 Effectiveness and performance-related testing results

Table 4.1 Performance-related results

Table 4.2 Timing results of object cloning and state restoration

Table 5.1 Run-time overheads imposed by reflective operation calls

Table 5.2 Pre- and post-conditions of CA action `LoadPress1`

Table 5.3 Failure modes of robot and press 1 and failure detection

Table 5.4 Two examples of exceptional post-conditions

Table 5.5 Exceptional post-conditions as to concurrent failure

List of Programs

Program 3.1 Class `SFTClass`

Program 3.2 Implementation of $t/(n-1)$-VP

Program 3.3 Implementation of SCOP

Program 4.1 The `CAaction` class

Program 5.1 The AR class

Program 5.2 The `Controller` class

Program 5.3 The `add` operation

Program 5.4 Implementation of the actual control

Program 5.5 Interface of CA action `LoadPress1`

Program 5.6 Body of CA action `LoadPress1`

Program 5.7 The `Press1` controller

Program 5.8 The `LoadPress1` action

Chapter 1

Introduction

1.1 Motivation

The information age is now beginning. Computers are permeating our modern society and improving the quality of our daily lives. However, as the requirements for and dependences on computers increase, the likelihood of crises caused by computer failures, directly or indirectly, also increase! The consequences of these failures might be just inconvenience, e.g. incorrect billing, missed airline or hotel reservations etc., but in certain application areas they could be large economic losses, e.g. interruptions of banking systems or even loss of human life, e.g. failures of air, rail, and subway control systems. Although the root causes of computer failures may be physical, design (typically software), human-machine interaction faults, or even malicious attacks, software faults are becoming a major source of reported outages and system failures [Gray 1990] as software becomes more and more complex. Even for control applications that have less complex software, it is already well established that many failures are, in fact, caused by software bugs [Hecht & Hecht 1996].

The "software crisis" issue came to the fore in the late 1960s, for example through the discussions in the 1968 and 1969 NATO Software Engineering Conferences. The concept of software engineering, aimed at minimizing the risk of software failures, has offered many improvements in the way software is produced and in its quality. Such improvements can be achieved by the use of *fault avoidance* techniques, including structured programming, software reuse, and formal methods, to prevent software faults, and by the use of *fault removal* techniques, including testing, verification, and validation to detect and remove software faults. Unfortunately, no matter how rigorously fault avoidance and fault removal techniques are applied, software faults often escape the software development process and enter the field. In reality, even the best quality software systems experience 1 ~ 2 faults per 20,000 lines of uncommented code [Lyu 1995]. When we are not able to somehow produce fault-free software, it is rather necessary to investigate some alternative, or complementary techniques, including, for example, *fault tolerance* techniques that attempt to increase reliability by designing software to continue to provide service despite the bugs that the software may still contain.

Over the last two decades, there has been a considerable amount of research, as well as practical software engineering, in the area of software fault tolerance, (Chapter Two will provide a comprehensive overview of the current state of the art and state of practice for building fault-tolerant software). Unlike hardware failures that are, for the most part, due to physical degradation, most software faults are the result of software specification and design mistakes, and thus simple replication of software components does not provide appropriate protection. Techniques for tolerating software faults usually require design redundancy, that is, production of two or more software components is aimed at delivering the same service through independent designs and realizations. Although there are different experiences and some doubts about design redundancy and related techniques, academia and industry appear to have reached a general consensus that *fault tolerance techniques for coping with software faults, if used properly, have the capability of increasing the reliability of a computer-based system* [McAllister & Vouk 1996].

However, existing techniques, especially redundancy-based schemes, are not yet sufficiently mature as a practical engineering discipline for routine applications. A number of conceptual and practical issues are still open or not solved satisfactorily. Though we are clearly aware that fault-tolerant software does improve system reliability, with the current level of understanding it is difficult to precisely know how much system reliability will be actually increased by such software in practice. This leads to the question of whether the use of design redundancy will provide a sufficient reliability improvement for critical systems that require very high software reliability. A further question is how cost-effective fault-tolerant software will be, assuming the desired reliability could be achieved.

Concurrent programs, such as operating systems and real-time control systems, are usually extremely complex. As compared with sequential programs, there is a stronger rationale for incorporating fault tolerance into such systems to improve their reliability. Unfortunately, common fault tolerance mechanisms for sequential systems cannot be simply applied to concurrent programs where many new technical and practical problems arise, and some of the problems are still not well understood. Even simple exception handling, rather than redundancy-based methods, for concurrent software is still an evolving subject and clear consensus has not yet been reached [Cristian 1995]. Many design concepts, such as *conversations* [Randell 1975], are well studied and developed, but the practice has not been well established for the majority of realistic concurrent systems, so that design concepts often have to be adjusted and recalibrated when they are considered for actual languages and systems.

Although much is understood about software fault tolerance, techniques for building fault-tolerant software still require further investigation and better understanding. Transferring key techniques and methodologies in software fault tolerance from an art to a routine-based practice is still be a major challenge now and likely to remain well into the next millennium.

1.2 Goals of This Research

The first goal of this research is to develop some innovatory schemes for building fault-tolerant software and to study the ways of enhancing software reliability and improving the balance between reliability and efficiency, in comparison with existing techniques for tolerating software faults. (The proposed schemes are essentially intended for incorporating fault tolerance into sequential programs though they may be applied to concurrent systems under an appropriate framework for coping with concurrency.)

The second goal of this research is to study issues related to fault tolerance in concurrent software systems. With the aim of overcoming the mismatch between the development of pure design concepts and practical aspects of realistic languages and systems, an object-oriented environment is assumed to facilitate the discussion of several key practical issues, such as concurrent sharing of objects, effective prevention of erroneous information smuggling [Kim 1982] and coordinated exception handling.

System-level support for implementing fault-tolerant software is also important though they are a relatively weak part of the current research — no technique can become actually viable for normal usage if the application programmers have to take care of such additional, low-level details as software version synchronization and state restoration. The third goal of this research is to design and develop a supporting system architecture, using the latest pattern technique [Buschmann et al 1996], to alleviate the development effort of fault-tolerant software for both sequential and concurrent programs.

We are aware that the maturity of software fault tolerance techniques towards an engineering practice needs substantial, long-term, industrial and academic efforts. We hope that all the above goals as a whole represent a consistent attempt of bridging the gap between theoretical concepts and practical applications.

1.3 Thesis Overview

As taken by others in similar research, this thesis takes a system approach rather than a programming language approach since the major issues with software fault tolerance are rooted in system design and language can be regarded as an implementation tool for the system designer. The key language-related issues are discussed in object-oriented terms, starting with complex concurrent programs. More programming language issues and implementation issues are taken into account while addressing the implementation of a multi-level reference architecture and its associated architectural pattern.

In order to maintain good readability and also to demonstrate a clear track of how basic concepts could be enriched and extended to practical engineering techniques, this thesis is organized following two logical strings. One string starts, in Chapter Three, from simple, sequential program systems and then goes on to tackle more complex, concurrent program issues in Chapter Four. The other string first deals with pure design

concepts in Chapter Three, then turns to address in Chapter Four more practical issues in an assumed object-oriented environment, and, in Chapter Five, finally ends up with architectural patterns, actual concurrent programs and an industrial case study.

1.3.1 Main Results

The major results obtained from this research are as follows.

- A survey of state of the art techniques and state of practice approaches to software fault tolerance is given with an abundant bibliography that covers latest progress.

- Two new schemes are developed for building fault-tolerant software. The $t/(n-1)$-VP approach, i.e. $t/(n-1)$-*variant programming*, is aimed at increasing software reliability and controlling additional complexity. The SCOP approach, i.e. *self-configuring optimal programming*, presents an adaptive way of dynamically adjusting software reliability and efficiency aspects.

- Both dependability and efficiency improvements achieved by $t/(n-1)$-VP and SCOP are modelled and evaluated analytically, as compared with main existing schemes like recovery blocks and N-version programming. The analytic conclusions are also supported by experimental data and evaluation.

- *Coordinated Atomic (CA) actions* [Xu et al 1995a], as a general scheme for constructing fault-tolerant concurrent systems, is examined thoroughly. Key properties of this scheme, in particular enclosure (i.e. error confinement) and fault tolerance, are addressed in detail and formalized using a simplified logic system.

- A general model of exception handling, especially for concurrent/distributed object systems, is developed using CA actions as a system structuring unit. Both forward and backward error recovery are considered within the combined framework supported by exception handling and the CA action abstraction. New object diversity techniques are introduced to provide software fault tolerance in such concurrent/distributed systems.

- A multi-level system architecture for implementing fault-tolerant software is presented to help separate different concerns and impose desirable structuring characteristics. An architectural pattern is described in detail to tackle many practical issues. Both a reflective implementation and a delegation-based solution are discussed respectively with experimental data and the sample program code. Finally, a realistic industrial case study, the Fault-Tolerant Production Cell [Lötzbeyer 1996], is used to examine and confirm most of the ideas developed in this research.

1.3.2 Thesis Structure and Related Publications

The rest of this thesis is composed of five chapters.

Chapter Two attempts to give a comprehensive survey of the state of the art and the state of practice for building fault-tolerant software. This chapter is partially based on [Randell & Xu 1995] prepared for the Wiley book "Software Fault Tolerance" [Lyu 1995]. I undertook the detailed analysis of the various approaches to software fault tolerance, especially those developed outside Newcastle, that constituted the greater part of this paper.

Chapter Three develops two advanced techniques, $t/(n-1)$-VP and SCOP, addressing respectively the improvement of software reliability and the dynamic trade-off between dependability and efficiency. An early version of the $t/(n-1)$-VP approach appeared in *IEEE Trans. Reliability* [Xu & Randell 1997]. The SCOP approach was developed jointly with A. Bondavalli and F. Di Giandomenico [Xu et al 1995b]. My particular contributions to this work include the trade-off analysis, dependability evaluation and implementation of SCOP.

Chapter Four investigates the key issues related to fault tolerance in concurrent/distributed object systems. CA actions and exception handling are used to constitute a general framework for achieving fault tolerance in such complex systems. The concept of CA actions was first developed within the Dependability Group at Newcastle in 1995 [Xu et al 1995a]. Key properties of this scheme was formalized subsequently with our colleagues at University of Ulm, Germany, especially D. Schwier. My particular contributions to this formalization include enclosure and fault tolerance properties. The study of coordinated exception handling was performed jointly with A. Romanovsky and B. Randell [Xu et al 1998a]. My major contributions to this research are the conceptual model, correctness proofs, and complexity and performance analysis.

Chapter Five presents a multi-level system architecture, introduces an architectural pattern for implementing fault-tolerant software and conducts an industrial case study. The system architecture is developed partially based on our early work within the Dependability group in 1995 [Xu et al 1995c]. The task of experimental evaluation using C++ and Open C++ was undertaken subsequently together with B. Randell and A. Zorzo [Xu et al 1996]. I made particular contributions to the introduction of a generic framework, the design of experimental settings and the evaluation of software fault tolerance based on fault injection. Over the years I have been involved in a number of experiments and case studies conducted by ESPRIT projects PDCS, PDCS2 and DeVa, such as Production Cell I, II and III. I took the major responsibility of designing a control program for the Fault-Tolerant Production Cell (i.e. Production Cell II) [Xu et al 1998b], focusing on the development of various strategies for coping with software faults and hardware component failures.

Finally, Chapter Six concludes this thesis and discusses the way forward.

Paginated blank pages are scanned as found in original thesis

No information is missing

Chapter 2

Fault-Tolerant Software

An important method of coping with software design faults is through fault tolerance. Software is said to be fault-tolerant if it can continue to provide service despite the existence of residual faults after its development. The term "software fault tolerance", in the context of this thesis, is concerned with all the techniques necessary to enable a system to tolerate software faults, although the effectiveness of these techniques is not usually limited to a precise class of faults. In reality, transient hardware faults, hardware design faults and software bugs often cause similar system behaviour [Powell 1991].

There are a few survey papers and texts on the subject of tolerance to software faults, such as [Strigini 1990][Lyu 1995][McAllister & Vouk 1996][Hecht & Hecht 1996]. But most of them focused on sequential programs and few discuss the issues related to supporting mechanisms. The main goal of this chapter is to provide a broader overview of the techniques for building fault-tolerant software, covering the recent advances and problems in both sequential and concurrent programs and discussing the issues with system-level support for implementing fault-tolerant software.

Section 2.1 gives a taxonomy of terms and introduces a set of concepts which will be used frequently throughout the whole thesis. Section 2.2 reviews the fault tolerance techniques mainly for sequential programs. Section 2.3 addresses the issues concerned with fault tolerance in complex concurrent/distributed systems. Section 2.4 covers the subject that has received less attention in the literature — system-level support and environments. The last section gives a brief summary of this chapter.

2.1 Basic Terminology and Concepts

Before discussing basic principle of software fault tolerance and various existing techniques, we need to first introduce some fundamental concepts and outline an abstract model for software systems.

2.1.1 System Model

We are concerned with software systems, which may be implemented on a variety of hardware. A *system* consists of a number of *components*, which cooperate under the

control of a *design* to service the demands of the system *environment* [Anderson & Lee 1981]. The components and the system environment themselves may be viewed as systems in their right. The design can be also considered as a special component (or an algorithm) that is responsible for defining the interactions between components and establishing connections between components and the system environment.

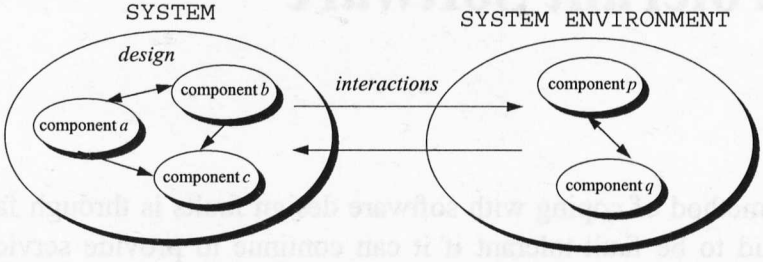


Figure 2.1 System and components

In actual software systems, components may be categorized as being either synchronously or asynchronously related to the design which employs them. Synchronous components are passive, they are invoked by their calling environment and will complete before the environment may resume. The top-down decomposition of a sequential program into a hierarchy of procedures is a typical example of design using synchronous components. By contrast, asynchronous components are active and, once invoked, will operate asynchronously with their environment. For example, communicating concurrent processes is a common form of program design utilizing asynchronous components.

Systems, and their components, can be regarded as performing operations in order to provide responses to requests. It is each component's responsibility to alert its environment when it cannot carry out a requested operation. In fact, with the system model of Figure 2.1, we need a clear discipline for *exception handling*. Apart from normal responses, there are two distinguished classes of exceptional responses: an *interface exception* is signalled when interface checks determine that an invalid service request has been made to a component and its environment that made the invalid request must deal with the exception; and a *failure exception* is the means by which the component notifies its environment that it has been unable to provide the service requested of it. In addition, within a component a *local exception* may also be raised when the component has detected an abnormal condition that its own *exception handlers* should deal with, so that if possible the component returns to its normal activities without affecting its environment.

Indeed, an *idealized fault-tolerant component* [Anderson & Lee 1981] should in general provide both normal and abnormal (i.e. exception) responses in the interface between interacting components, in a framework which minimizes the impact of these provisions on system complexity. In other words, such components should provide a means of system structuring which makes it easy to identify *what* parts of a system have *what* responsibilities for trying to cope with *which* sorts of fault [Randell 1984].

2.1.2 Dynamic Behaviour of Software Systems

Before we can clearly define the failures of a system, we need to further discuss the dynamic behaviour of software systems and, in particular, the sequence of the events that leads to the failure of these systems. In principle, the dynamic behaviour of a software system is characterized by the series of *internal states* which the system adopts during its processing. Certain elements of an internal state will coincide with the interface between the system and its environment, and these constitute the *external state* of the system via which its external behaviour can be realized. Each internal state will comprise the set of data values within the scope of the design — output values produced by the components (i.e. their external states) and the values of any variables maintained directly by the design. Under normal processing conditions, the system will advance from one valid internal state to the next by means of a valid transition (see Figure 2.2).

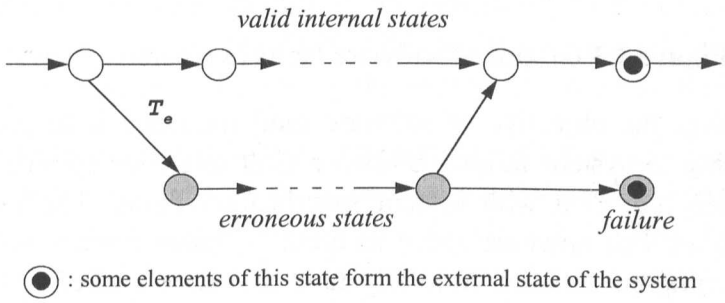


Figure 2.2 System's dynamic behaviour and erroneous transition T_e

2.1.3 Software Faults, Errors and Failures

When a *fault* is encountered in the software during its processing, an erroneous transition may occur which transforms the system to an invalid internal state containing one or more defective values or *errors*. If an error in an internal state maps on to the external state, for example when an incorrect value is output, then a *failure* of the system will result. Therefore, all system failures can be attributed to errors in the internal state of the system (but not all errors necessarily result in failure). All errors and, therefore, all failures are attributable to faults in the system. For a given system, its failure may be caused by the failure of the system design algorithm to perform its internal function, i.e. a *design fault*, or may derive from the failure of a system component to operate according to its specification (a component fault). Since a system will eventually decompose purely into a set of designs, all software faults can be considered as design faults at some level of abstraction within the software system.

2.1.4 Origins of Software Faults

Traditionally, the failure of a system is considered to occur when the external behaviour of the system first deviates from that defined in its specification. In reality,

the specification may contain errors due to incompleteness, inconsistencies and ambiguities, it may not accurately reflect the true requirements which the environment places on that system. Therefore, the behaviour which is correct with respect to the specification may be still viewed as a perceived failure in terms of the expectations of the environment. Figure 2.3 summarizes the major origins of software faults.

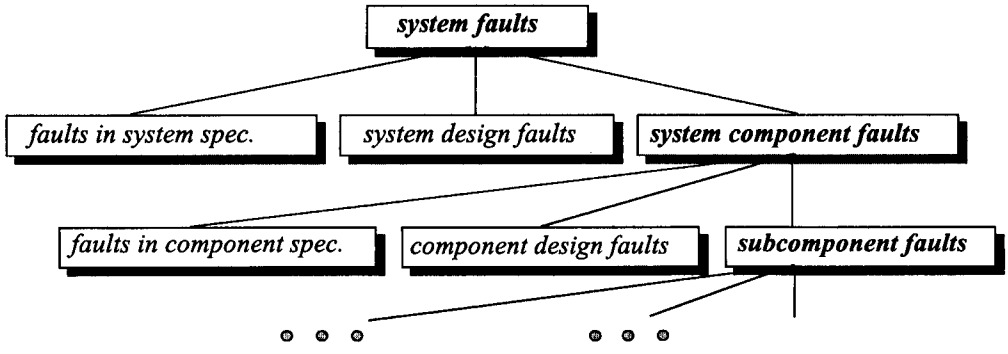


Figure 2.3 Origins of software faults (a recursive view)

Generally speaking, the objective of software fault tolerance is to prevent software faults from leading to system failure. Software fault tolerance applied to the system level will basically not deal with system specification faults. The use of the fault tolerance techniques will however strive to protect against system design faults and various system component faults. Similarly fault tolerance applied at the component level will normally not cope with erroneous component specifications. (To improve the correctness of system specifications, we may need to develop diverse specifications and use appropriate formal techniques [Avizienis 1985].)

2.1.5 Software Fault Tolerance

A general way of allowing a system to operate successfully in the presence of a design fault is to construct the entire system from a number of *diverse designs* (or *software redundancy*) derived from a common, presumably correct, specification [Randell 1975][Avizienis & Chen 1977]. These diverse designs should have a low probability of exhibiting common-mode failure, i.e. by producing similar erroneous output for the same processing conditions. There are four major activities common to any scheme for providing software fault tolerance:

1) *Error detection:* Faults are not directly detectable but the effects of a fault, namely one or more errors in the internal state of a system (or component), can be used to identify the presence of a fault. It is important to detect errors before they affect the external state of the system, namely cause system failure. Error detection includes various measures and mechanisms, such as executable assertions, voter of diverse outputs, and memory-protection mechanisms.

2) *Damage assessment and confinement:* Once the internal state of the system (or component) contains one or more errors, the extent of this damage must be

assessed. This is usually achieved by having damage-confinement structures present within the system that limit the propagation of errors.

Example 2.1: The *atomic action* concept is an important dynamic structuring concept that helps to confine damage. The activity of a group of components constitutes an atomic action if there are no interactions between that group and the rest of the system for the duration of the activity [Anderson & Lee 1981].

3) *Error recovery:* Once the extent of the damage to the internal state of the system (or component) is assessed, this damage must be repaired so that failure of the system can be averted. Error recovery can be *forward* — the system is returned to a further error-free state by applying corrections to the damaged state. Recovery can be also *backward* — the system is recovered to a previous error-free state.

4) *Fault treatment:* Faults remaining in the system after error recovery can lead to further errors. Removing the fault will usually require off-line diagnosis and repair. Components should be re-configured into the system after repair.

When multiple software versions of a common specification are considered, further classification of software faults, such as the terminology used in [Avizienis 1985], is useful. *Independent faults* in different versions usually cause *distinct* errors (although they may lead to similar errors by chance). *Related faults* result either from an erroneous specification, common to all the versions, or from dependences in the diverse designs and implementations. Related faults manifest under the form of *similar errors*. Similar errors can cause *common-mode failures* of multiple versions while distinct errors usually lead to *separate failures*. However, this classification should be treated as one kind of modelling assumption used by some researchers (e.g. [Laprie et al 1987][Arlat et al 1990]) and should not be regarded as a description of reality.

For certain applications the detection of inability to deliver acceptable results may be crucial. The failure classification defined in [Laprie et al 1987] is based on such consideration: *detected failures* indicate no acceptable results found and no results delivered; *undetected failures* imply that erroneous results are delivered.

2.1.6 Various Measures

Software *reliability* is defined as the probability of failure-free software operation for a specified period of time in a specified environment [ANSI 1991]. This definition can be applied equivalently to system level and component levels. Software reliability is in fact one of the attributes of software *quality*, a multidimensional property including other customer satisfaction factors such as functionality, usability, performance, and maintainability [Grady 1992]. But software reliability is generally accepted as the key factor in software quality since it quantifies software failures.

There is often confusion between the concept of reliability and of *availability*. The availability of a system can be stated as the probability that the system will satisfy a request for a service. A 0.999999 availability means the software system is not available at most one hour in million hours. An increase in reliability will improve the availability of a system but the inverse is not necessarily true. A system with high availability may in fact fail, but its recovery time and failure frequency must be small enough to achieve the required availability.

Safety is another related but different attribute of software quality, emphasizing the non-occurrence of catastrophic consequences on the environment of the software. [Leveson 1986] details the difference and defines safety as the probability that conditions that can lead to hazards, do not occur in a specified time, regardless of the functioning of the system. Again, a reliable system can be unsafe.

A well-known obstacle to a discussion of system reliability and techniques for fault tolerance is the lack of agreed terminology for the relevant concepts. To overcome some of these difficulties, *dependability* is an appropriate generic term and defined as the quality of the delivered service such that reliance can justifiably be placed on this service [Laprie 1992]. It describes a general attribute of a system that encompasses measures such as reliability, availability, safety and security. A terminology tree from [Laprie 1992] is reproduced in Figure 2.4.

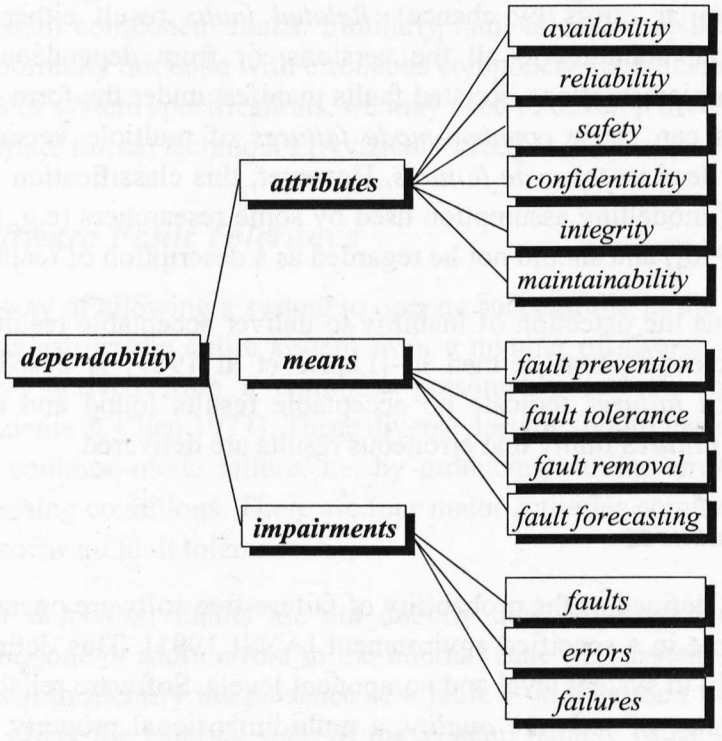


Figure 2.4 The dependability tree

2.1.7 Cost, Effectiveness and Efficiency

Besides the technical factors, there are also economical factors that affect the development of fault-tolerant software [Knight & Ammann 1991]. The *cost* issue of multi-version software has to be considered carefully. Although there are some experiments that show multi-version software is capable of reducing failure probability over the single versions from which they are built, the systems being compared in such experiments were usually not built with equal cost.

A comparison of the cost of building a fault-tolerant system with the cost of building a non-fault-tolerant system often becomes essential. To examine the *cost effectiveness* of fault-tolerant software properly and precisely, it is necessary to build and operate both a fault-tolerant system and a non-fault-tolerant system using comparable resources. Indeed, for a more precise calculation, the cost comparison will further depend upon whether costs are calculated at the end of development or at the end of operational life because multi-version software may require extra maintenance efforts.

Closely related to cost considerations, software *efficiency* is also considered as one of the attributes of software quality and defined as the good use of hardware resources, such as processors and communication devices, during the software execution.

2.1.8 System Support for Fault-Tolerant Software

Since fault-tolerant software requires the addition of software redundancy to normal software, some extended syntax and certain run-time support may be needed. If the application programmers were responsible for treating many low-level details such as state saving/restoring and message passing between software versions, implementation and maintenance of fault-tolerant programs would become very complex. The objective of a *supporting system* or environment for developing fault-tolerant software is to make the development a standard, and as much as possible programmer-transparent, activity. Figure 2.5 shows an example which is similar to the supporting environment developed in [Ancona et al 1990].

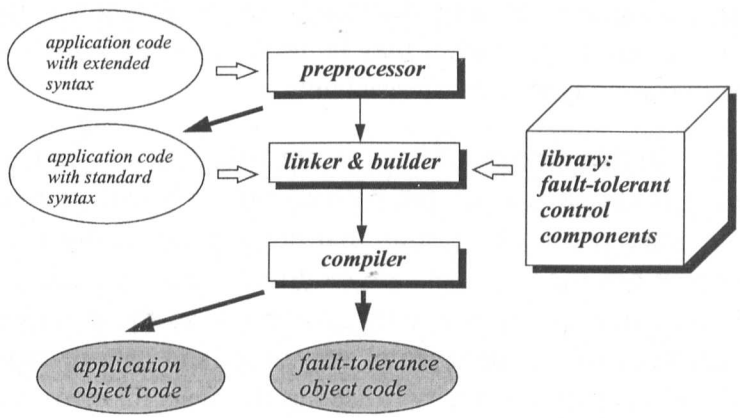


Figure 2.5 An example of supporting systems

2.2 Fault Tolerance in Sequential Software

In this section, we mainly discuss the schemes used in sequential programs, although the essence of these schemes can be extended and applied to concurrent software (this will be discussed in Section 2.3).

2.2.1 Techniques for Dealing with Software Faults

As discussed in Section 2.1, software faults are almost exclusively design- and implementation-related. However, some software faults may be caused by undetected hardware errors such as transient or timing faults, which often occur because of complex hardware/software/operating system interaction. Such faults (called *Heisenbugs* in [Gray 1990]) can be rarely duplicated or diagnosed.

Proposals for software fault tolerance started to appear in the mid 1970s [Elmendorf 1972][Horning et al 1974][Avizienis 1975][Fischler et al 1975][Kopetz 1974][Randell 1975]. From then on, some methodological proposals have been defined. Most proposals, such as recovery blocks (RB) [Horning et al 1974] and *N*-version programming (NVP) [Avizienis & Chen 1977], suggest to have independent programmers produce functionally equivalent software components, namely to use the design diversity approach.

> *Design diversity:* The production of two or more components is aimed at delivering the same service through independent designs and realizations [Avizienis 1985]. The components, produced through the design diversity approach from a common service specification, are called *variants* (alternates in RB or versions in NVP.) By incorporating at least two variants of a system, tolerance to design faults necessitates an *adjudicator* [Anderson 1986] (a decision algorithm) that provides an (assumed to be) error-free result from the execution of variant(s).

The diversity idea can also be used in the data domain [Ammann & Knight 1988] as well as in the operating environment domain [Gray & Siewiorek 1991][Huang & Kintala 1995]. The resulting approaches may be useful in some specific situations and be able to cope with certain types of software faults.

> *Data diversity:* In this approach, if a program fails with its original data, the same program is executed again but with slightly different, or re-expressed data. This is based on an observation that programs typically fail on some special inputs. When the input data are modified a little, the failure may not occur. However, this approach is only appropriate to those applications in which the accuracy of the input is not very strict and data re-expression is possible. (Sometimes data diversity is used as the basis for error detection as well, e.g. by relying on continuity properties of the function implemented.)

As mentioned at the beginning of this section, the failures exhibited by software faults can be transient, namely, the failures may not recur if a program is re-executed even on the same input, but after a certain amount of clean-up and reinitialization. This is because the behaviour of a program depends upon not only the input data and message contents but also upon timing factors, interleaving of different messages, shared variables and other state values involved in the operating environment of the application [Huang & Kintala 1995]. (For example, the timing factor is exploited for developing a time-diversity-based approach in the current Hitachi experiment [Kanekawa et al 1998].)

> *Environment diversity:* This approach is mainly aimed at software faults that cause transient errors. If a program fails, the program is rolled back to a previously valid state and is executed again with certain changes in its operating environment, such as a different hardware processor and the re-arrangement of message order. Such a pragmatic approach seems to fit practical experience, but it is less complete and its effectiveness may be a matter of luck.

There are some limited form of software fault tolerance; for example, by detecting and recovering an error, and either ignoring the operation which generated it or by providing a pre-defined and heavily degraded service. In such cases software cannot be regarded as truly fault-tolerant since some perceived departure from specification is likely to occur. However, this approach can result in software which is *robust* in the sense that catastrophic failure can be averted. Robust software is discussed usually in the context of exception handling facilities [Cristian 1984].

Finally, robust data structures [Taylor et al 1980][Taylor & Black 1985] are another example of coping with special classes of software faults, namely, software faults that manifest themselves by corrupting data structures. Such techniques attempt to protect the structural information of data structures by adding redundancy in their representation in storage.

2.2.2 Basic Schemes: Recovery Blocks and N-Version Programming

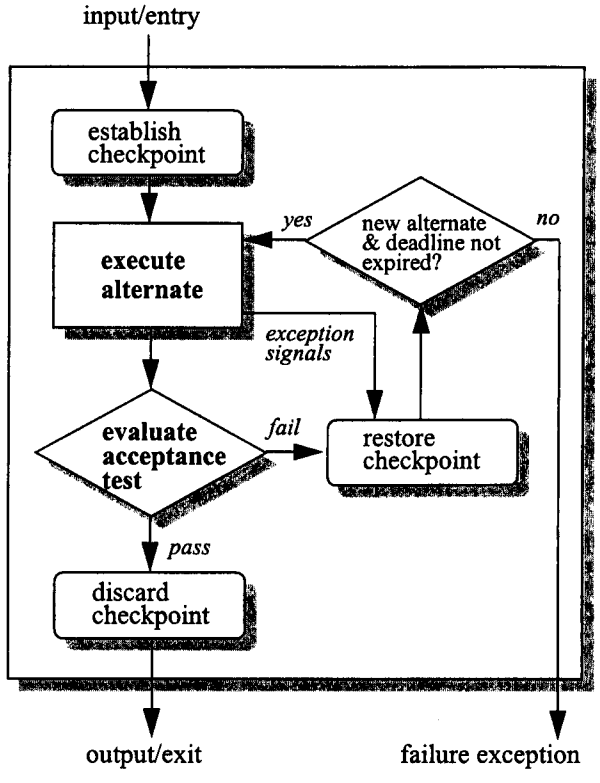
This section will concentrate on the two main comprehensive software fault tolerance schemes, i.e. recovery blocks and *N*-version programming, both of which include diverse designs in software for fault treatment. Conceptually, the methods the two schemes use to tolerate faults are different. Recovery blocks uses backward error recovery that tries to return the system to a previous, error-free state from which execution may be re-tried with a new variant. *N*-version programming uses forward error recovery to construct a valid, error-free new state from existing redundant information, provided by multiple software variants.

Recovery Blocks

A research effort into the recovery block scheme was started in the early 70s by Brian Randell and his research group [Horning et al 1974][Randell 1975]. The basic recovery block was developed for sequential systems. (Details of extensions for use in concurrent systems are discussed in Section 2.3.) The recovery block approach attempts to prevent residual software faults from impacting on the system environment, and it is aimed at providing fault-tolerant functional components which may be nested within a sequential program. Figure 2.7(a) shows the usual recovery block syntax and (b) gives an architectural view of the scheme and its operation.

```
ensure      acceptance test
by          primary alternate
else by     alternate 2
            .
            .
else by     alternate n
else error
```

(a)



(b)

Figure 2.6 Recovery block syntax and its operation

On entry to a recovery block, the current state of any variables that might be changed during the execution of the recovery block must be saved in a secure storage area called a *recovery cache* [Horning et al 1974][Anderson & Kerr 1976] to permit backward

error recovery, i.e. establish a checkpoint. The primary alternate is executed and then the acceptance test is evaluated to provide an adjudication on the outcome of this primary alternate. If the acceptance test is passed then the outcome is regarded as successful and the recovery block can be exited, discarding the information on the state of the system taken on entry. However, if the test fails or if any errors are detected by other means during the execution of the alternate, then an exception is raised and backward error recovery is invoked. This restores the state of the system to what it was on entry. After such recovery, the next alternate is executed and then the acceptance test is applied again. This sequence continues until either an acceptance test is passed or all alternates have failed the acceptance test. If all the alternates either fail the test or result in an exception (due to an internal error being detected), a failure exception will be signalled to the environment of the recovery block. Since recovery blocks can be nested, then the raising of such an exception from an inner recovery block would invoke recovery in the enclosing block.

The overall success of the recovery block scheme rests to a great extent on the effectiveness of the error detection mechanisms used — especially (but not solely) the acceptance test. The acceptance test must be simple otherwise there will be a significant chance that it will itself contain design faults, and so fail to detect some errors, and/or falsely identify some conditions as being erroneous. Moreover, the test will introduce a run-time overhead which could be unacceptable if it is very complex. The development of simple, effective acceptance tests can thus be a difficult task, depending on the actual specification. In practice the acceptance test in a recovery block should be regarded as a last line of detecting errors, rather than the sole means of error detection. The expectation is that it will be buttressed by executable assertion statements within the alternates and run-time checks supported by the hardware. Generally, any such exception raised during the execution of an alternate will lead to the same recovery action as for acceptance test failure.

As described in [Melliar-Smith & Randell 1977] forward error recovery can be further incorporated into recovery blocks to complement the underlying backward error recovery. (In fact, a forward error recovery mechanism can support the implementation of backward error recovery by transforming unexpected errors into default error conditions [Cristian 1982].) If, for example, a real-time program communicated with its (unrecoverable) environment from within a recovery block then, if recovery were invoked, the environment would not be able to recover along with the program and the system would be left in an inconsistent state. In this case, forward recovery would help return the system to a consistent state by sending the environment a message informing it to disregard previous output from the program.

In the first paper about recovery blocks [Horning et al 1974], Horning et al list four possible failure conditions for an alternate: 1) failure of the acceptance test, 2) failure to terminate, detected by a timeout, 3) implicit error detection (for example divide by zero), and 4) failure exception of an inner recovery block. Although the mechanism for implementing the time-out detection measure was not discussed by the authors, the

original definition of recovery blocks does cover this issue. Several implementations of watchdog timers for recovery blocks have been described in [Hecht 1976][Kim & Welch 1989]. Timeout can be also provided as a syntactic form in the recovery block structure [Gregory & Knight 1985].

Although each of the alternates within a recovery block endeavours to satisfy the same acceptance test there is no requirement that they all must produce the same results [Lee 1978]. The only constraint is that the results must be acceptable — as determined by the test. Thus, while the primary alternate should attempt to produce the desired outcome, the further alternate may only attempt to provide a degraded service. This is particularly useful in real-time systems, since there may be insufficient time available for fully-functional alternates to be executed when a fault is encountered. An extreme corresponds to a recovery block which contains a primary module and a null alternate [Anderson & Knight 1983][Anderson et al 1985]. Under these conditions, the role of the recovery block is simply to detect and recover from errors by ignoring the operation where the fault manifested itself.

N-Version Programming

Algirdas Avizienis and his research group started in 1976 a research effort into the *N*-version programming approach [Avizienis & Chen 1977][Chen & Avizienis 1978]. *N*-version programming provides run-time fault tolerance by comparing the outputs produced by several diverse software versions (i.e. variants) and tries to mask version failures by propagating only consensus results. The dimension of diversity was originally based on independent programmers, but more recent studies have extended the dimension by using a mixture of diverse formal specifications [Avizienis & Kelly 1984]), diverse design (e.g. different algorithms and data structures), different programming teams, and diverse implementations (e.g. different languages, tools or compilers).

The NVP approach is a direct application of the hardware *N*-modular redundancy approach (NMR) to software. *N* versions (i.e. variants) of a program that have been independently designed are executed in parallel and their results compared by a decision mechanism. By incorporating a majority vote, the system can eliminate erroneous results (i.e. the minority) and pass on the (presumed to be correct) results (i.e. the majority). The execution of *N* variants is supposed to take advantage of the redundant hardware processors likely to be available in a system that must tolerate both hardware and software faults. In the situation that hardware resources are not sufficient, software variants may be executed sequentially. Grnarov, Arlat and Avizienis [Grnarov et al 1980] sketched such a sequential application of NVP, called NVS. Figure 2.7 illustrates NVP's architecture and its operation. (There is no general syntax proposal for NVP.)

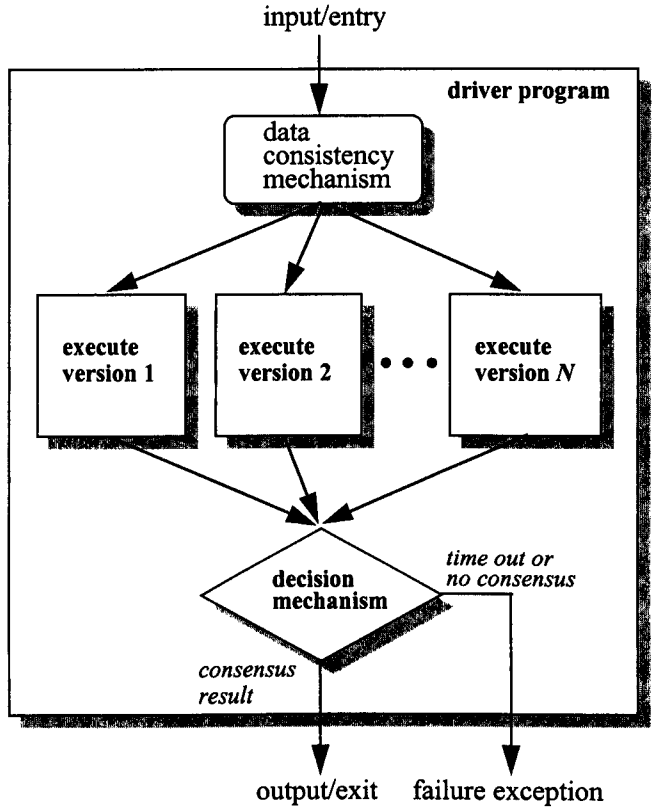


Figure 2.7 NVP architecture

Conceptually NVP is very simple — one of the scheme's advantages. A driver program (or main program) is responsible for 1) invoking each variant with the identical input data, 2) waiting for all variants to complete, and 3) executing the decision mechanism to determine a consensus result. Coordination of each of the variants is based on simple synchronization primitives. There can be no interaction between variants and they must be prevented from making global state changes and direct outputs.

The success of NVP critically depends upon the decision mechanism that identifies the erroneous output of a faulty variant. This requires that special attention be placed on the design of the decision mechanism, which may itself be diverse. There are two main functions of the decision mechanism: i) to collect the available results from the software variants and select the consensus result to be delivered to the user, and ii) to diagnose the cause of any detected errors and perform the appropriate error recovery. In order to increase the effectiveness of error detection and decision, it is normal practice to include in the specification of the variants intermediate cross-check values which will be delivered to the driver program together with the output results. Using these cross-check values, the voter can detect certain errors in the internal states of the variants as well as in their external states. However, this may have a negative impact on the degree of design independence of the various variants.

In many complex cases, the simple majority vote must be combined with more sophisticated and application-oriented tests. For example, the performance and fault

coverage of the decision mechanism can be increased by adding an acceptance test to each variant to exclude clearly erroneous results from the consensus decision. These tests can serve as a form of filter [Anderson 1986] to prevent incorrect results from being sent to the decision mechanism or to prevent the mechanism from waiting for a result that will never arrive (e.g. using a timeout-based test).

In practice the variants may deliver their results to the decision mechanism at markedly different times. An individual result may be delayed due to hardware processor interrupts, internal error detection and recovery, or the use of diverse algorithms. Since the decision mechanism usually requires the collection of all results from the software variants before it can determine a consensus, a slow variant would delay the decision and in some cases the decision cannot be made in the pre-specified amount of time. This problem will be more serious if real-time applications are taken into account. A watchdog timer may be used to ensure that the decision is made and the consensus result delivered in a timely manner.

Another complication occurs in the design of the voting check when the results involve non-discrete values such as real numbers, since different variants may produce slightly differing correct results. The problem is often regarded as a form of replica non-determinism. To allow a consensus to be reached despite these differences requires inexact voting. This requirement for inexact voting can lead to application dependent algorithms and hence loss of generality. ([Di Giandomenico & Strigini 1990] gave a comprehensive survey of various voters and adjudicators for diverse redundant components.)

Comparison of RB and NVP

There is a detailed discussion of relative advantages and disadvantages of recovery blocks and *N*-version programming in [Lee & Anderson 1990]. In general, the critical importance of effective error detection in software fault tolerance would suggest that *N*-version programming is most appropriate if for a given application voting checks may be easily implemented and replicated hardware is available to reduce run-time overheads. However, the recovery block scheme is a generally applicable approach which can be mapped naturally onto nested component structures and would be most appropriate for those systems where hardware resources are limited and voting checks is inappropriate.

A brief comparison of the two schemes is given in Table 2.1, summarizing major differences between them. In many aspects, recovery blocks and *N*-version programming are complementary according to their relative merits. The final choice of the scheme depends very much upon the characteristics of an actual application and the support hardware on which it will run. In order to combine various features of each scheme, many hybrid approaches have been developed and we will review some typical and advanced schemes in the next subsection.

Characteristic Aspects	Recovery Blocks	*N*-Version Programming
error detection	By acceptance testing	by result comparison
Execution of variants	Conditionally sequential execution of alternates with respect to detected errors Single processor required	Parallel execution of *N* versions multiple processors required
Adjudication	Acceptance testing (absolute test)	voting and majority decision (relative test)
fault treatment	new alternate following backward recovery (dynamic redundancy)	fault masking by ignoring erroneous results (static redundancy)
support for diverse designs	degraded designs possible but global data structure required	local data structure possible but no degradation and cross-check points may be required
Structural overheads	An acceptance test and extra alternates recovery cache	a voter and extra versions data consistency and synchronization mechanisms
run-time overheads	Less predictable; execution of alternates and the acceptance test triggered by detected errors	execution time of the slowest version and duration of voting
Concurrent system recovery	Conversation-type schemes and variations (see review in Section 2.3)	lack of related research (see work in Section 4.4)

Table 2.1 Comparison of RB and NVP

2.2.3 Advanced Schemes

Many applications and varieties of recovery blocks, *N*-version programming, and their combinations have been explored and developed by various researchers. Some of the most typical extensions and advanced schemes are considered below.

Distributed Recovery Blocks

H. Hecht was the first to propose the application of recovery blocks to flight control systems [Hecht 1976][Hecht & Hecht 1986]. His work included an implementation of a watchdog timer that monitors availability of output within a specified time interval and his model also incorporates a rudimentary system to be used when all alternates of the recovery block scheme are exhausted. Since then, further researches and experiments have been conducted by Hecht and his colleagues. For example, M. Hecht et al [Hecht et al 1989][Hecht et al 1991] described a distributed fault-tolerant architecture, called

the Extended Distributed Recovery Block, for nuclear reactor control and safety functions. Their architecture relies on commercially available components and thus allows for continuous and inexpensive system enhancement. The fault injection experiments during the development process demonstrate that the system could tolerate most single faults and dual faults.

K. H. Kim and his colleagues in the DREAM Laboratory have extensively explored the concept of distributed execution of recovery blocks, a combination of both distributed processing and recovery blocks, as an approach for uniform treatment of hardware and software faults [Kim 1984][Kim & Welch 1989][Kim & Yoon 1988][Welch 1983]. The details are given in [Kim 1995]. A useful feature of their approach is the relatively low run-time overhead it requires so that it is suitable for incorporation into real-time systems. The basic structure of the distributed recovery block is straightforward: the entire recovery block, two alternates with an acceptance test, is fully replicated on the primary and backup hardware nodes. However, the roles of the two alternate modules are not the same in the two nodes. The primary node uses the first alternate as the primary initially, whereas the backup node uses the second alternate as the initial primary (see Figure 2.8). Outside of the distributed recovery block, forward recovery can be achieved in effect; but the node affected by a fault must invoke backward recovery by executing an alternate for data consistency with the other nodes. To test the execution efficiency of the approach, two experimental implementations and measurements have been conducted on distributed computer networks [Kim & Min 1991][Kim 1993]. The results indicate the feasibility of attaining fault tolerance in a broad range of real-time applications by means of the distributed recovery blocks.

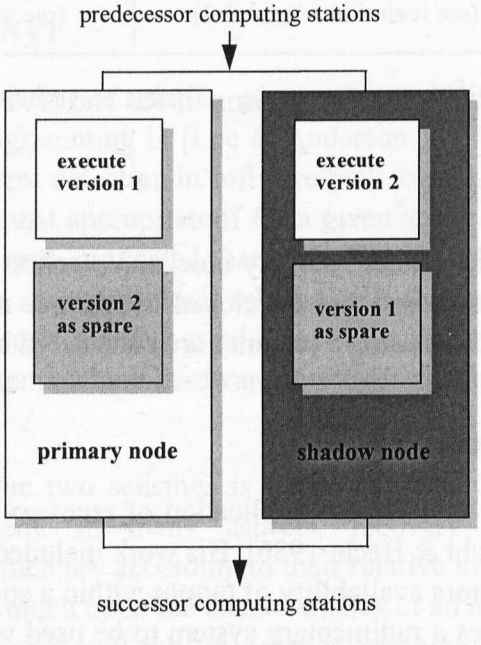


Figure 2.8 DRB architecture

N Self-Checking Programming

As discussed previously, recovery blocks can be equivalent to stand-by sparing in hardware and *N*-version programming can correspond to *N*-modular redundancy. Laprie et al [Laprie et al 1987] attempted to identify the equivalent to active dynamic redundancy used in hardware. They investigated some existing industrial practice and described it as *N* Self-Checking Programming (NSCP).

NSCP consists of *N* self-checking components designed independently, but with the same functionality. A self-checking software component is considered as resulting either from the association of an acceptance test with a variant, or from the association of two variants with a comparison algorithm. Fault tolerance can be provided by the parallel execution of $N(\geq 2)$ self-checking components. During the execution, a self-checking component is regarded as being active; the others are considered as "hot" spares. Upon the failure of the active component, service delivery is switched to a self-checking component previously regarded as a spare. Error processing is thus performed through error detection, by an acceptance test or a comparison algorithm, and switching of the results.

In reality, some of real-life systems such as the Airbus A-320 and the Swedish railways' interlocking system do not actually employ RB or NVP, but use, perhaps relatively simple, self-checking software.

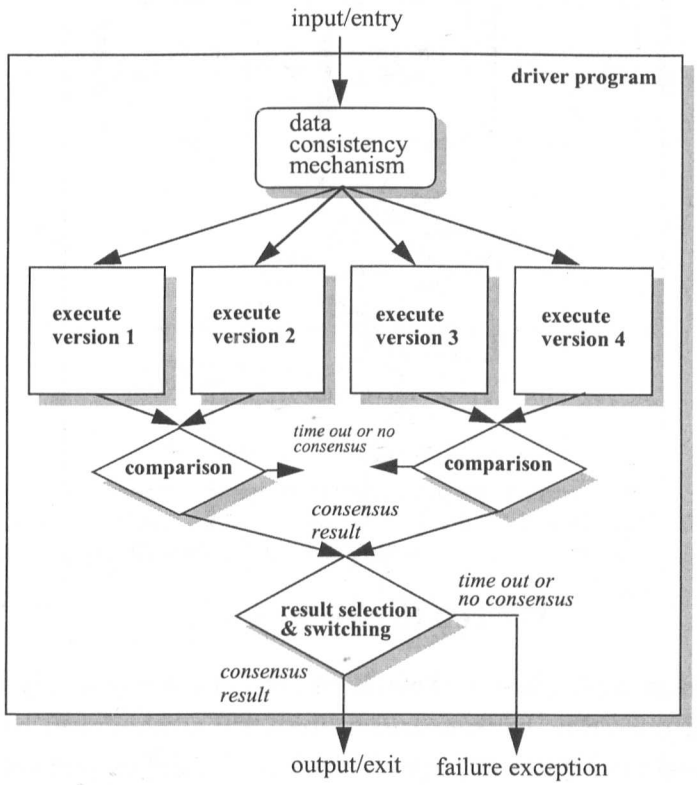


Figure 2.9 NSCP architecture ($N = 4$)

Consensus Recovery Blocks

The consensus recovery block (CRB) [Scott et al 1985] is an attempt to combine the techniques used in recovery blocks and *N*-version programming. It is claimed that the CRB technique reduces the importance of the acceptance test used in the recovery block and is able to handle the case where NVP would not be appropriate since there are multiple correct outputs. The CRB requires design and implementation of *N* variants of the algorithm which are ranked (as in the recovery block) in the order of service and reliance. On invocation, all variants are executed and their results submitted to an adjudicator, i.e. a voter (as used in *N*-version programming). The CRB compares pairs of results for compatibility. If two results are the same then the result is used as the output. If no pair can be found then the results of the variant with the highest ranking are submitted to an acceptance test. If this fails then the next variant is selected. This continues until all variants are exhausted or one passes the acceptance test.

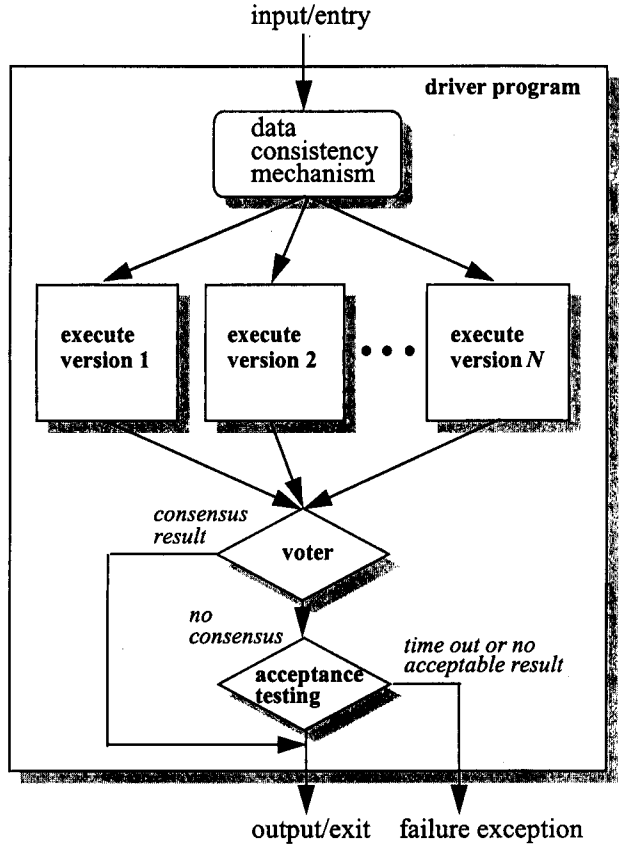


Figure 2.10 CRB architecture

Reliability models are developed in [Scott et al 1987] for the recovery block, *N*-version programming and the CRB. In comparison, the CRB is shown to be superior to the other two. However, the CRB is largely based on the assumption that there are no common faults between the variants. (This is not totally true according to the experiments in [Knight et al 1985][Scott et al 1984].) In particular, if a matching pair is found, there is no indication that the result is submitted to the acceptance test, so a

correlated failure in two variants could result in an erroneous output and would cause a catastrophic failure.

The converse of the CRB scheme, proposed by [Athavale 1989][Gantenbein 1991][Belli & Jedrzejowiez 1991], is called acceptance voting (AV). As in NVP, all variants execute in parallel. The output of each variant is then presented to an acceptance test. If the acceptance test accepts the output it is then passed to a voter. The voter processes only those outputs which have passed by the acceptance test. Since the voter may not have the same number of outputs at each invocation, the voting algorithm must be dynamic with respect to the number of acceptable outputs.

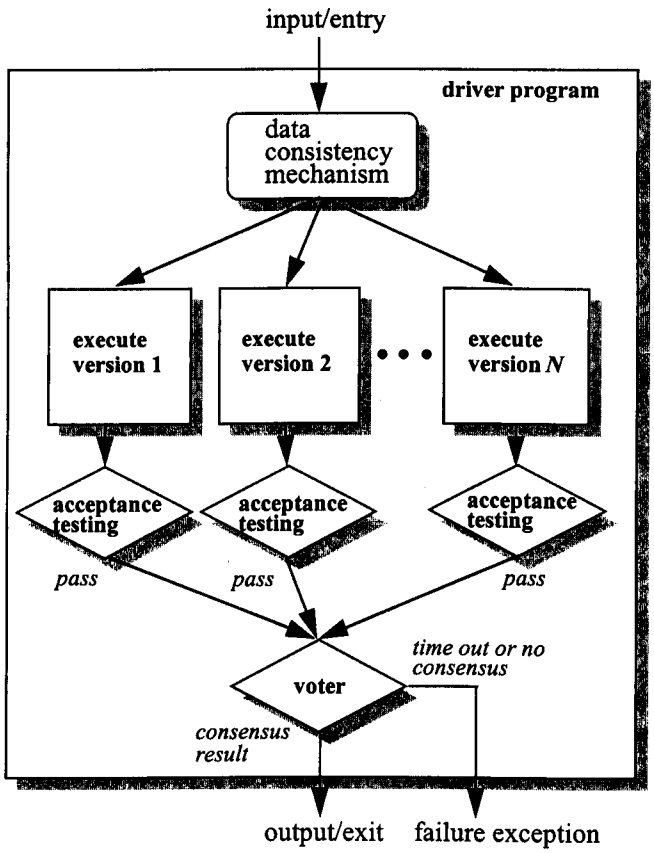


Figure 2.11 AV architecture

The voting itself can be made adaptive as well. A generalization of majority voting is consensus voting (CV) [McAllister et al 1990]. The CV scheme first seeks for an absolute majority; if it fails to find the majority, it tries to identify a relative majority (i.e. less than $\lceil (N + 1)/2 \rceil$). When there are more than one such majority, CV either selects one randomly, in the NVP case, or applies them to an acceptance test, for the CRB architecture. The CV strategy is shown particularly effective in small output space because it automatically adjusts the voting to the changes in the effective output space cardinality.

Other Schemes and Techniques

There are also other schemes and variations of the basic schemes. We will briefly discuss two of them below.

The Maximum Likelihood Voting (MLV) method is recently proposed as one of the voting methods with high reliability [Leung 1995]. This method determines the most likely correct result from *N* software variants based on the reliability history of each variant. Assuming that output space is finite and that the variant failures are statistically independent, MLV employs the likelihood value of each variant output to select an answer as a correct output. The original proposal of MLV assumes failure independence. [Kim et al 1996] examines MLV under the failure correlation conditions and concludes that MLV outperforms statistically several implementations of NVP, RB and CRB.

Sullivan and Masson developed an algorithm-oriented scheme, based on the use of what they term Certification Trails [Sullivan & Masson 1990][Sullivan & Masson 1991]. The central idea of their method is to execute an algorithm so that it leaves behind a trail of data (certification trail) and, by using this data, to execute another algorithm for solving the same problem more quickly. The outputs of the two executions are compared and considered correct only if they agree. An issue with the data trail is that the first algorithm may propagate an error to the second algorithm, and this could result in an erroneous output. Nevertheless, the scheme is an interesting alternative to the recovery block scheme, despite being perhaps of somewhat limited applicability.

2.2.4 Dependability and Cost Effectiveness

Fault-tolerant software techniques, especially those based on diverse designs, have the capability of achieving an improvement over non-fault-tolerant software. Practical experiences (see Sections 2.2.5 and 2.2.6) are more positive than negative, though a number of issues remain open and controversial. Typical questions include items such as: how much can fault-tolerant software actually increase system dependability in practice, and how cost-effective is software fault tolerance?

The potential for common-mode failures among the software variants designed independently is the major reason for these questions. Experiments show that incidence of common-mode failures of variants in fault-tolerant software may not be negligible in the context of current software development and testing techniques [Scott et al 1984][Vouk et al 1985][Knight & Leveson 1986a][Kelly et al 1988][Eckhardt et al 1991]. However, the origins and the extent of common-mode failures in practical systems are still not well understood. [Lyu & He 1993] conjectured that improving software development process could increase overall system dependability effectively.

To resolve the above issues, it is essential for us to have some measurable or predictable confidence in fault-tolerant software's operation. There are two basic ways

to achieve this: 1) direct measurement of the software dependability, and 2) the use of an analytic model to predict the dependability. The usual approach to direct measurement is *life testing* — a system is exercised in its operational environment (or a laboratory approximation of its environment), and the failures of the system over time are observed. The number of failures observed over time can be then used to derive an estimate of the probability of failure and to establish a confidence level in that estimate. Unfortunately, the amount of testing for high dependability levels that are typically required by critical systems is infeasible [Miller 1989][Littlewood & Strigini 1993]. Unlike hardware testing, there is no general way of testing in an unbiased manner that can shorten the observation time required to assess the dependability of fault-tolerant software [Knight & Ammann 1991].

The other way of assessing the software dependability is to use an analytic model to derive predictions of system dependability from measurements of component dependability. For multi-version software, quantitative dependability modelling generally depends upon knowledge of the common-mode failure probability distribution. Unfortunately, this distribution is not known in general, making the dependability of fault-tolerant software difficult to assess. We shall further discuss the analytic evaluation of fault-tolerant software in the next section.

If an adequate level of dependability cannot be easily ensured by the design of multiple versions, the cost may become a problem. The frequently asked question is: what dependability would be achieved if the same cost as a multi-version system were spent on the quality of a single version system? Although there is no a general answer to this, there is practical evidence that in some cases the design diversity approach is cost-effective. Table 2.2 reports some empirical results and analytic conclusions.

Experimental Systems or Analytic Model	Number of Software Variants	Extra Cost of Fault-Tolerant Software
Ericsson [Hagelin 1987]	two	< 100%
Experiment in [Panzl 1981]	two	77%
PODS [Bishop et al 1986]	Three	126%
naval C^2 system [Anderson et al 1985]	two	60%
analytic cost model [Laprie et al 1990]	an additional variant	75 ~ 80%

Table 2.2 Extra cost of fault-tolerant software

The experience of the Ericsson company demonstrated that the development cost of two software variants is not double the cost of a single software version [Hagelin 1987]. Panzl [Panzl 1981] found that dual-program development effectively reduced the number of software errors from 69 to 2 with an extra development cost of 77%. The

cost of 3-version programming in the PODS project [Bishop et al 1986] was estimated as about 126% of single version costs. Of the nine residual faults detected, only two faults were common between versions, and all other fault combinations were effectively masked by the voting mechanism. Another large project was set up at Newcastle University at the early 80s to evaluate the cost effectiveness of applying recovery blocks to a naval command and control demonstrator. The results of experimentation on the completed demonstrator revealed that 74% of potential failures were averted by the recovery block scheme. The increased development cost was approximately 60%.

Laprie et al [Laprie et al 1990][Laprie et al 1995] established a simple analytic model for calculating the cost of fault-tolerant software. A number of sources of increasing costs, such as synchronization, failure detection and adjudication, are considered. Several factors that reduce the cost of per version are also taken into account, such as back-to-back testing. By their abstract cost model, the cost of *N*-version software is less than *N* times of the cost of a single version. In a multi-version setting a typical version cost is about 75 to 80 percent of a normal single version costs.

The cost of fault-tolerant software has to be treated carefully. Knight and Ammann [Knight & Ammann 1991] argued that, for a cost comparison to be valid, it is necessary to built and operate both a multi-version system and a single-version system using comparable resources, and to calculate the costs at the end of operational life rather than at the end of development phases. Following their arguments, the total cost of *N*-version software would be greater than *N* times the cost of a single software version. If multi-version software cannot generally achieve a large improvement in dependability, can the high cost be justified in practice? We can only answer this if we know the cost of failure! Typical cases include consumer electronics where recall costs are extraordinarily high, and fly-by-wire aircraft (see Section 2.2.6) in which software development costs only make up a small part of the overall cost.

Les Hatton [Hatton 1997] has conducted a detailed analysis based on the well-known Knight and Leveson experiment [Knight & Leveson 1986a][Knight & Leveson 1986b][Brilliant et al 1990], whose results were originally used to "demonstrate" the impracticality of multi-version software. Hatton instead used these results, together with other evidence, to argue that multi-version development techniques can produce more dependable systems than concentrating all effort into producing one "good" version. This is because, with the current state of the art and the current state of practice in software engineering, we are not able to actually make one really good version no matter how much effort we expend. Although the dependability gain of *N*-version software is much less than that which would in theory be gained by *N* completely independent versions without possible common-mode failure, the gain is still substantial: Hatton's study shows that the average (dependability) improvement obtained by a 3-version system was by a factor of 5 to 9 in comparison with a single "best" version.

In summary, multi-version high quality software can offer more dependability than we can gain any other way. Although there are certain difficulties in directly measuring or accurately predicting the dependability of multi-version software, and the cost of such software can be very high, the analysis based on empirical results suggests that the multi-version development techniques are significantly superior to even the current state of the art regarding all other approaches to high software dependability, especially in situations where cost of failure is high.

2.2.5 *Analytical Evaluation and Experimental Validation*

Over the years techniques for fault-tolerant software have been evaluated by a number of researchers and engineers, both theoretically (e.g. [Scott et al 1984][Arlat et al 1988][Tomek et al 1993][Dugan & Lyu 1995]) and experimentally (e.g. [Anderson & Kerr 1976][Avizienis et al 1988a][Eckhardt et al 1991][Lyu & He 1993].)

Analytical Evaluation

Analytical estimations of effectiveness of fault-tolerant software provide an alternate approach when direct measurement of very high dependability is infeasible. Analytic modelling of different fault tolerance schemes can give insight into their behaviour and allows quantification of their relative merits. The major difficulty of using existing models is that of estimating the values of the parameters used in the models. It is particularly difficult to predict the probabilities of errors common to software variants. Intuition suggests that such probabilities should be made as small as possible, and the related values should be obtained experimentally. Unfortunately, given the current state of the art and practice we are not able to determine such parameter values with any high confidence.

The early models usually assumed statistical independence of failures (or a slightly weaker assumption) in fault-tolerant software. Scott et al [Scott et al 1984] were first to develop models to treat common-mode failures, but their models become quite complicated and intractable as the number of software versions increases for the general case. Eckhardt and Lee [Eckhardt & Lee 1985] analyzed the effect of common-mode failures on the performance of *N*-version software using a model that incorporates the observation that certain inputs are more likely to cause failure than others. Based on this model, the reliability of a multi-version system can be predicted if the probability distribution associated with common-mode failure is known. [Eckhardt et al 1991] shows that, using very limited empirical evidence (which may have no general implication) for such probability distributions, only a modest improvement factor of 2 to 5 in comparison with a single version under certain conditions can be achieved by the use of the multi-version software technique. Littlewood and Miller extended and generalized the Eckhardt and Lee model by addressing the effect of different development strategies for each of the *N* version [Littlewood & Miller 1989]. They found that disjoint failures of multiple versions result in even better performance

(if such a result can be obtained in practice.) Popov and Strigini [Popov & Strigini 1998] further extended the previous conceptual models in [Eckhardt & Lee 1985][Littlewood & Miller 1989][Nicola & Goyal 1990] to improve the understanding of the various ways failure dependence between versions can arise. Their analysis provides some useful insight into non-intuitive aspects of the failure process of multi-version software.

In the late 80's, Arlat et al [Arlat et al 1990] modelled the behaviour of a software system as a Markov chain, and provided dependability modelling and detailed evaluation of recovery blocks and *N*-version programming. The major contributions of their work to the area of analytic modelling include 1) the definition of a modeling framework based on the identification of possible types of faults through the analysis of software production process — an analytic method discussed in detail in [Laprie 1984], 2) the evaluation of both reliability and safety, and 3) the detailed analysis of two specific architectures: nested RB and NVP with a failed version. Their modelling framework is subsequently extended for a performability analysis of fault-tolerant software techniques in [Tai et al 1993] and for a dependability analysis under a distributed computing environment in [Di Giandomenico et al 1997].

The assumptions regarding common-mode software failures in different versions are different in various existing models. The model developed in [Arlat et al 1990] assumes that similar errors are caused by related software faults and different errors which are simultaneously activated are caused by independent faults. Related and independent faults are assumed to be mutually exclusive. However, [Dugan & Lyu 1995] assumes that these two types of faults are statistically independent. [Dugan 1994] gave a detailed comparison of several typical modelling approaches.

There are only a few papers that have considered a combined analysis of fault-tolerant software and hardware [Laprie et al 1987][Di Giandomenico et al 1997]. Laprie et al [Laprie et al 1990] conducted a dependability analysis of hardware and software fault-tolerant architectures adopting a Markov approach. Three special architectures that tolerate a single hardware or software fault were examined in detail. [Dugan & Lyu 1995] used a combination of fault tree and Markov modelling as a framework for the analysis of hardware and software fault tolerant system. When a Markov model is used to represent the effects of permanent hardware faults, a fault tree model is used to capture the effects of software faults and transient hardware faults. Such a hierarchical modelling approach can simplify the development, solution and understanding of the modelling process.

It is important that practical fault-tolerant systems are analyzed not only with respect to their data-domain characteristics, but also with respect to their time-domain characteristics. Tomek et al [Tomek et al 1993] modelled recovery blocks with failure correlation and analyzed the time-dependent behaviour of RB reliability in considerable detail. The categories of different events developed in Pucci's model [Pucci 1992] were used to establish the stochastic reward nets (SRNs) for recovery

blocks. Kanoun et al [Kanoun et al 1993] have modelled reliability growth of individual components using the time-dependent hyperexponential model. They applied the model to the analysis of reliability of RB and NVP and found that NVP is much more sensitive to the removal of independent faults at the testing stage than RB since the failures that did occur are most possibly correlated among different versions after most independent faults have been removed.

Experimental Work

This section discusses diverse software experiments conducted in different universities, especially those using recovery blocks, *N*-version programming and distributed recovery blocks. Experiments involved in industrial applications will be discussed in Section 2.2.6).

The first implementation of recovery blocks involved defining and simulating a simple stack-oriented instruction set, incorporating a recovery cache [Anderson & Kerr 1976]. Simple test programs embodying recovery blocks could be run on this machine simulator, and have deliberate faults injected into them. Visitors to the project were typically challenged to try and cause a demonstration recovery block program to fail — their inability to do so was a persuasive argument for the potential of the recovery block scheme! Another experimental system is described in [Shrivastava 1978][Shrivastava & Akinpelu 1978] in which recovery blocks were incorporated in the language Pascal. The modification was made to the kernel and interpreter of Brinch Hansen's Pascal system to support the syntax of recovery blocks and the associated recovery caches needed for state restoration. For experimental sample programs, the run-time overhead ranged between 1 to about 11% of T1 (execution time of a program without any recovery facilities) when no errors are detected. If a primary failed, the time taken to restore system state was up to about 30% of T1. This experiment also showed that recovery caches made a substantial saving in space, compared with complete checkpointing. In order to further enhance the performance of recovery blocks, the next major work at Newcastle on the implementation of the basic recovery block scheme involved the design and building of a hardware recovery cache for the PDP-11 family of machines [Lee et al 1980].

The controversial nature of software fault tolerance spurred extensive efforts aimed at providing evidence of the scheme's potential cost-effectiveness in real systems. (The developers of *N*-version programming [Avizienis & Chen 1977] were similarly motivated to undertake extensive experimental evaluations, as discussed later.) During 1981-84 therefore, a major project directed by Tom Anderson applied an extension of recovery blocks in the implementation of a Naval Command and Control system composed of about 8000 lines of CORAL programming, and made use of the above-mentioned hardware cache [Anderson et al 1985]. The practical development work included the design and implementation of a virtual machine which supported recovery blocks, together with extensions to the CORAL programming language to allow software fault-tolerance applications to be written in this high-level language. Analysis

of experimental runs of this system showed that a failure coverage of over 70% was achieved. The supplementary cost of developing the fault-tolerant software was put at 60% of the implementation cost. The system overheads were measured at 33% extra code memory, 35% extra data memory and 40% additional run time. These led to the conclusion that "by means of software fault tolerance a significant and worthwhile improvement in reliability can be achieved at acceptable cost" [Anderson et al 1985].

Delta-4 was a collaborative project carried out within the framework of the European Strategic Programme for Research in Information Technology (ESPRIT) [Powell 1991]. Its aim was the definition and design of an open, dependable, distributed computer system architecture. [Barrett & Speirs 1993] describes the integration of software fault tolerance mechanisms into the existing Delta-4 architecture. The authors claimed that the incorporation of recovery blocks and dialogues (structures for supporting inter-process recovery) into the Delta-4 framework is obtained without significant overheads.

Experimental evaluations of *N*-version programming have been mainly performed at UCLA. The first experiment at UCLA was a small program developed for the numerical solution of partial different equations [Avizienis & Chen 1977][Chen & Avizienis 1978]. Eighteen programmers implemented the program using three different algorithms. Of the 71 cases in which a single version failed, 59 were successfully masked while 12 cases caused the system to abort. The most significant cause of the common-mode failures was found to be coincident omission of certain key results. The initial experiment was followed by an experiment which investigated the impact of specification on residual faults [Kelly & Avizienis 1983][Avizienis & Kelly 1984]. Based on three specifications written in English, OBJ and PDL respectively, eighteen PL-1 programs were developed for a database application. This experiment demonstrated a large increase in dependability in the multi-version executions, and thereby showed the feasibility of design diversity. One of the following research activities has been concerned with the development of a test bed for *N*-version experiments [Avizienis et al 1985][Avizienis et al 1988a], and its use to investigate issues such as inexact voting strategies and forward recovery of failed versions.

The second generation experiment in fault-tolerant software at UCLA was sponsored by the NASA Langley Research Center, also involving several other U.S. universities and research centres [Kelly et al 1986][Kelly et al 1988]. A specification of an avionic application was supplied to 20 teams. Software versions were developed in Pascal and ranged in size between 2000 and 5000 lines of code. The analysis of the twenty versions has provided some insight into the causes of common-mode failure. The experiment has again shown the effectiveness of *N*-version programming in tolerating the faults introduced during the design and coding phases of development. Another experiment conducted by UCLA and Honeywell-Sperry Commercial Flight Systems Division was to develop and evaluate six versions of a flight control program in six different programming languages [Avizienis et al 1988b]. Preliminary results showed

the effectiveness of using different languages to enforce diversity — few faults were found to be common to more than one version.

Scott and his colleagues at North Carolina State University and other institutions [Scott et al 1984] were first to show, using an experiment, that programs may not fail independently, and they developed models to treat this issue. A large-scale experiment was conducted by Knight and Leveson [Knight & Leveson 1986a] to further investigate the issues regarding the independence of versions and common-mode failures. A 27-version program was developed and submitted to one million test patterns, encountering common-mode failures of two or more versions on 1255 occasions. This was regarded statistically as demonstrating that the failures of versions were not always independent. Some doubts have been raised about several aspects of the experiment, such as random test cases without using any enforced diversity, questionable granularity of the decision vector, and the inappropriate application which is not complex enough to permit true diversity. In a later experiment [Knight & Leveson 1986b] the average probability of failure for a three-version system randomly constructed from the 27 versions was found to be 19 times less than the average for individual versions, showing that significant reliability improvements can still be achieved. The later NASA-LaRC study [Eckhardt et al 1991][Vouk et al 1993] investigated a large number of instances of common-mode failures among 20 versions, and found that significant reliability improvements may not be always ensured; in some cases, only minor improvements were observed. Lyu and He [Lyu & He 1993] considered three and five version configurations formed from 12 different versions and demonstrated how the improved development process could increase the overall dependability of multi-version software.

Since the initial formulation of the distributed recovery block (DRB) concept in 1983 [Kim 1984], several experiments were conducted, including the application of the DRB scheme to adjacent computing stations in real-time parallel processing multi-computer testbeds [Kim & Welch 1989][Kim & Min 1991] and to LAN based systems [Hecht et al 1989][Hecht et al 1991][Kim et al 1994]. These experiments essentially demonstrated the DRB scheme to be a practical technique that could be used in many real-time applications with acceptable amount of time overheads.

2.2.6 Industrial Applications

Fault-tolerant software based on design diversity, though not yet widely used in general-purpose industrial areas, has been attempted for use in a number of critical application areas. For example, in nuclear power plants [Gmeiner & Voges 1979][Bishop et al 1986], in railway systems [Hagelin 1987][Kantz & Koza 1995], and in aerospace systems [Avizienis et al 1988b][Davis et al 1993]. More on the use of software diversity in computer-based systems can be found in two books [Voges 1987][Lyu 1995].

Nuclear Power Plants

Most application examples in nuclear industries have been experimental. Some experiments were conducted to examine the usefulness of dual programming for back-to-back testing [Ramamoorthy et al 1981][Dahll & Lathi 1979], and other experiments were related to *N*-version programming, including the project run by the Kernforschungszentrum in Karlsruhe [Voges 1987] and the PODS project [Bishop et al 1986]. The PODS project had three diverse teams in England, Finland and Norway implementing a simple nuclear reactor protection system application. With good quality control and experienced programmers no design-related faults were found when the diverse programs were tested back-to-back. All the faults were caused by omissions and ambiguities in the requirements specification. However, because of the differences in interpretation between the programmers, five of the faults occurred in a single version only, and just two common faults were found in two versions.

There are at least two known examples involving the practical utilization: 1) use of dual programming techniques in the Candu Plants [Popovic et al 1986] and 2) the protection of the Darlington nuclear reactor based on the diverse software concept [Condor & Hinton 1988]. The UK's Sizewell B nuclear reactor primary-protection system is the other practical example, which has three separate channels. However, the design diversity used in the current Sizewell B system is hardware-based only since each channel uses the same control software, as discussed in [Hatton 1997].

Railway Systems

The first industrial use of fault-tolerant software, based on the design diversity principle, is believed to have occurred in a railway system [Sterner 1978] reported in [Voges 1987]. Design diversity has been used to either help develop, verify, or actually implement an operational railway application for deployment in Sweden, Denmark, Finland, Switzerland, Turkey and Bulgaria [Hagelin 1987], Austria [Theuretzbacher 1986][Erb 1989], Italy [Frullini & Lazzari 1984], Singapore, and the United States [Turner et al 1987]. Such applications are still increasing. Several recent examples are the ELEKTRA system [Kantz & Koza 1995], the Shinkansen Train Control system [Hachiga et al 1993], and the SACEM system [Hennebert & Guiho 1993].

In 1985, Alcatel Austria started with the development of the electronic railway interlocking system ELEKTRA [Erb 1989]. A two channel system based on design diversity has been developed, and high availability and reliability are achieved by using actively triplicated redundancy with on-line recovery. In 1989, the first system was put into operation. Currently, about fifteen railway interlocking systems are in operation and further installations are ongoing [Kantz & Koza 1995].

Aerospace Applications

The use of fault-tolerant software in aerospace applications has received a lot of attention over the years, in both civilian [Sweet 1995] and military [Martin 1982][Turner et al 1987] aircraft, and in the U.S. space shuttle.

The slat/flap control system for the civilian Airbus A310 airliner consists of two functionally identical computers with diverse hardware and software [Hills 1983][Wright 1986]. The later Airbus-320 fly-by-wire flight control system has two types of computers and four software variants [Traverse 1987]. Four software variants are organized as two self-checking pairs to provide degradable services.

The Boeing 737-300 incorporates the Sperry SP-300 digital autopilot flight director system which is essentially a 2-version system with diverse hardware and software [Yount 1986]. The Sperry Corporation has also developed a prototype diverse system for the yaw damper of the Boeing 757 and 767 aircraft.

The resident back-up software (REBUS) was developed by Draper Labs at the early 80s as a flight experiment on the F-8 digital fly-by-wire flight control system for NASA [Slivinski et al 1984]. The experimental flights have had no problem so far and the basic concept has been chosen for the X-Wing and F-16 programs.

The NASA space shuttle [Madden & Rone 1984][Spector & Gifford 1984] carries a configuration of four computers, each loaded with the same software, to tolerate hardware faults, but there is also a fifth computer developed by a different manufacturer and running dissimilar software, which is executed only when the software in the other four computers cannot reach consensus during critical phases of the flight. The applicability of design diversity to the European Space Shuttle has also been investigated [Laprie et al 1987].

Other Known Examples

There are two well-known examples of systems coping with software faults without the use of diverse programs: the Electronic Switching Systems of the Bell Labs [Haugk et al 1985] and the Tandem Systems [Gray 1986]. Although both systems were mainly designed to tolerate hardware faults, their fault tolerance mechanisms are proved to be quite effective to handle and tolerate many software faults. In practice, a large percentage of remaining bugs in operational software can be "Heisenbugs" (transient) rather than "Bohrbugs" (solid) [Gray & Siewiorek 1991][Sullivan & Chillarege 1991]. Errors caused by Heisenbugs can rarely be duplicated or diagnosed. A common solution is to re-execute the software in the hope that the transient disturbance is over, perhaps after a certain amount of clean-up and reinitialization.

2.3 Fault Tolerance in Concurrent Software

Concurrent and distributed systems often give rise to complex asynchronous and interacting activities. On the one hand, the provision of fault tolerance becomes a very difficult task in such circumstances. On the other hand, these systems are very prone to errors and failures due to their extreme complexity. Fault tolerance is one of the practical methods for improving their dependability. However, most mechanisms for fault tolerance in sequential systems cannot be simply applied to concurrent systems in which many new technical problems arise. The existing techniques (e.g. those discussed in Section 2.2) must be adjusted and extended to cope with complex concurrent activities.

2.3.1 Models of Constructing Concurrent Fault-Tolerant Software

Many realistic applications, typically concerned with process control, avionics and telephone switching systems, are structured as concurrent processes communicating via messages. Fault tolerance in such systems usually is introduced through a controlled use of checkpoints by processes and a strict enclosure of communication between processes. The best-known scheme is the *conversation* scheme developed at Newcastle in early 1970s [Randell 1975]. Shrivastava et al [Shrivastava et al 1993] refer to this way of structuring an application as employing the *process-conversation* model (PM).

Another widely used technique for introducing fault tolerance, particularly in distributed systems, is based on the use of atomic actions (atomic transactions) that operate on objects for structuring programs. The class of applications where such an *object-action* model (OM) [Shrivastava et al 1993] has found usage include transaction processing applications in office information, airline reservation and database systems.

The conversation concept facilitates failure atomicity and backward recovery in cooperating process systems in a manner analogous to that of the atomic action mechanism in object systems. This terminological distinction between the area of communicating process systems and that of object-based systems is, [Shrivastava et al 1993] claimed, of only surface importance, namely PM and OM are dual of each other.

In the following we first review two models and the related developments respectively, address the specific aspects of the two models, and then argue the need for some combined and integrated models. In fact, the major difference of the two models is the kinds of concurrency they attempt to control and handle. In complex computing systems, there are at least three kinds of inter-process concurrency [Hoare 1978]. *Independent* concurrency means concurrent processes have access to only disjoint object sets, without any form of sharing or interacting. *Competitive* concurrency implies that concurrent processes compete for some common objects, but without explicit cooperation. *Cooperative* concurrency occurs in many actual systems, e.g. real-time control applications, where concurrent processes cooperate and interact with each other in pursuit of some joint goal; each process is responsible only for a part of the

joint goal. Independent concurrency is a trivial case to which all the fault-tolerant techniques for sequential programs can be applied directly. However, other kinds of concurrency must be controlled carefully either by underlying support mechanisms or by the application programmers in order to ensure consistent system states and facilitate error recovery. While OM focuses mainly on competitive concurrency, PM was developed originally for dealing with cooperative concurrency.

2.3.2 *Process-Oriented Systems and Conversations*

When a system of cooperating processes employs recovery blocks, each process will be continually establishing and discarding checkpoints, and may also need to restore the state to a previously established checkpoint. However, if recovery and communication operations are not performed in a coordinated fashion, then the rollback of a process can result in a cascade of rollbacks that could push all the processes back to their beginnings — the *domino effect* [Randell 1975]. This simply causes the loss of entire computation performed prior to the detection of the error. Figure 2.12 illustrates the domino effect with two communicating processes.

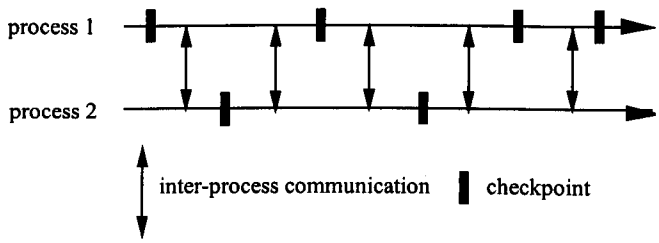


Figure 2.12 The domino effect

The conversation scheme provides a means of coordinating the recovery blocks of interacting processes to avoid the domino effect. Figure 2.13 shows an example where three processes communicate within a conversation and process 1 and process 2 communicate within a nested conversation. Communication can only take place between processes that are participating in a conversation together.

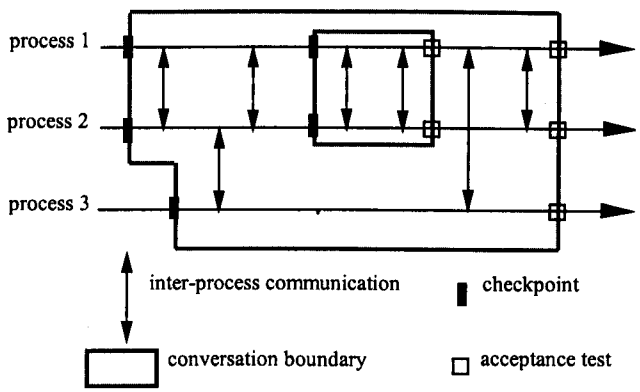


Figure 2.13 Nested conversations

The operation of a conversation is: 1) on entry to a conversation a process establishes a checkpoint, 2) if an error is detected by any process then all the participating processes must restore their checkpoints, 3) after restoration all processes use their next alternates, and 4) all processes leave the conversation together.

Considerable research has been undertaken into the subject of concurrent error recovery, including improvements on the conversation and different implementations of it. There are at least two classes of approaches to preventing the domino effect: the coordination-by-programmer approach and the coordination-by-machine approach. With the first approach, the application programmer is responsible for designing processes so that they establish checkpoints in a well coordinated manner [Randell 1975][Marshall 1980][Russell 1980][Kim 1982]. Many authors have added language constructs to facilitate the definition of restorable actions based on this approach, such as [Russell & Tiedeman 1979][Anderson & Knight 1983][Gregory & Knight 1985][Jalote & Campbell 1984][Jalote & Campbell 1986]. In contrast, the coordination-by-machine approach relies on an "intelligent" underlying processor system which automatically establishes appropriate checkpoints of interacting processes [Kim 1978][Barigazzi & Strigini 1983][Koo & Toueg 1987][Kim & You 1990]. If restorable actions are unplanned, so that the recovery mechanism must search for a consistent set of checkpoints, such actions would be expensive and difficult to implement. However, such exploratory techniques have the advantage that no restrictions are placed on inter-process communication and that a general mechanism could be applied to many different systems [Merlin & Randell 1978][Wood 1981]. To reduce synchronization delays introduced by controlled recovery, some researches have focused on the improvement of performance, such as the lookahead scheme and the pseudo-recovery block [Kim et al 1976][Kim & Yang 1988][Ramanathan & Shin 1988][Russell & Tiedeman 1979][Shin & Lee 1984]. (A few researches were also made into error recovery among the particular sets of so-called competing processes where the processes communicate only for resource sharing [Shrivastava & Banatre 1978][Shrivastava 1979].)

The original description of conversations provided a structuring or design concept without any suggested syntax. [Russell & Tiedeman 1979] proposed a syntax called the name-linked recovery block for the concept of conversations. Kim [Kim 1982] was the first to address in depth different syntactic forms for conversations based on the monitor structure. The different implementations deal with the distribution of the code for the recovery blocks of individual processes. The trade-off is either to spread the conversation among the individual processes such that all of the code of each process is in one location or have all the code for the conversation in one location.

There was no provision for linked forward error recovery in the original conversation scheme. Campbell and Randell [Campbell & Randell 1986] proposed techniques for structuring forward error recovery measures in asynchronous systems and generalized ideas of atomic actions so as to support fault-tolerant interactions between processes. A resolution scheme is used to combine multiple exceptions into a single exception if

they are raised at the same or nearly same time. Issarny extended their work to concurrent object-oriented systems by defining an exception handling mechanism for parallel object-oriented programming [Issarny 1993a]. This mechanism was then generalized to support both forward and backward error recovery [Issarny 1993b]. Also following the proposal in [Campbell & Randell 1986], Jalote and Campbell described a system which contains both forward and backward error recovery within a conversation structure (also known as an FT-Action). Their system was based on communicating sequential processes (CSP) [Hoare 1978] with one extension (the `exit`) statement. The declaration and general form of an FT-Action for a particular process *P* are given in Figure 2.14.

```
A: FT-Action with (P1, ..., Pn)
...
P1::[ ...
        FT-Action A
                <code>
                exit unless <e>
                <code>
                exit unless <e>
                ...
        end
```

Figure 2.14 An FT-Action

Forward error recovery in an FT-action is achieved through linked exception handlers where each process has its own handler for each exception. When an exception is raised by a process it is propagated to all the participating processes within the FT-action. Each process then executes its own handler for that exception. Backward recovery within an FT-action is obtained by recovery blocks. The `<code>` part in Figure 2.14 can be used to describe one of the primary and alternate modules of the recovery block scheme with various primitives that support acceptance testing and state restoration. Every participating process is required to have the same number of alternates. An FT-action can combine the two schemes in some limited forms so that forward and backward error recovery are used within the same structure. It can also cope with the issue of real-time applications through a simple timer.

Real-time applications may suffer from the possibility of *deserters* in a conversation — if a deadline is to be met then a process that fails to enter the conversation or to reach its acceptance test could cause all the processes in the conversation to miss that deadline [Kim 1982]. Russell and Tiedeman [Russell & Tiedeman 1979] considered relaxing the requirement for all processes exiting together so as to enable some protection against deserter processes, but this could lead to the domino effect. Campbell, Horton and Belford [Campbell et al 1979] proposed a deadline mechanism for dealing with timing faults. Anderson and Knight [Anderson & Knight 1983] proposed *exchanges* as a simplification of conversations where the cyclic nature of real-time systems is exploited. An exchange is a conversation in which all participating processes enter upon initiation and terminate upon exit. Error recovery is particularly

easy as the recovery data is only that needed upon initiation, which should only be a small amount of frame dependent data.

Gregory and Knight [Gregory & Knight 1985] identified a set of problems associated with conversations. They argued that there ought to be two types of acceptance test — one for each process within a conversation to check its own goal and one for the whole structure of the conversation to check the global goal. In addition, within a conversation or other structures mentioned above the set of processes that attempt their primary alternate is the same as the set of processes which attempt all other alternates, i.e. they all roll back and try again with their further alternates. This is overly restrictive and affects independence of algorithm between alternates. In an effort to solve these problems, the authors developed two concepts — a *colloquy* that contains many *dialogs*.

A dialog is a way of enclosing a set of processes in an atomic action. It provides no retry method and no definition of the action to be taken upon failure. If any failure occurs, the dialog restores all checkpoints and fails, signalling the failure to the surrounding colloquy. A colloquy that contains a set of dialogs controls the execution of dialogs and decides on the recovery action to be taken if the dialog fails (see Figure 2.15). The colloquy provides a means of constructing alternates using a potentially different set of processes, thereby permitting true diverse design. The dialog and colloquy allow time constraints to be specified and are accompanied by syntactic proposals that are extensions to the Ada language.

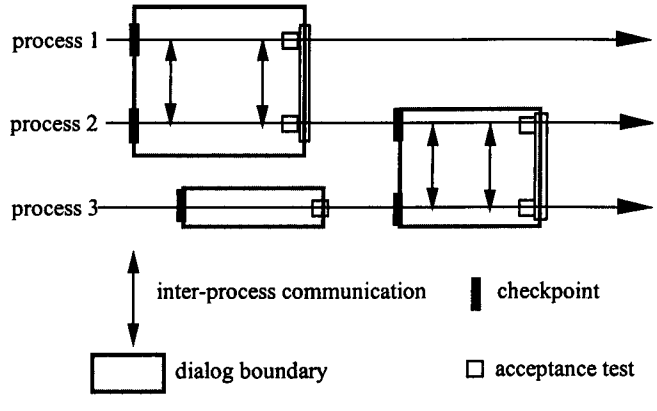


Figure 2.15 Three processes in a colloquy of three dialogs

However, when attempting the integration of the syntax for the colloquy into Ada, the authors found several new and potentially serious difficulties which arise because of a conflict between the semantics of modern programming languages and the needs of concurrent backward recovery [Gregory & Knight 1989]. The practical problems fall into the general categories of 1) program structure, 2) shared objects, and 3) process manipulation. All the problems have the potential to allow the state outside a dialog (or a conversation) to be contaminated by changes inside the dialog, i.e. *information smuggling*. Given the complexity and subtlety of these problems, Gregory and Knight

concluded that "the only workable solution might be that programming language design begin with backward error recovery as its starting point." Nevertheless, some preliminary and partial solutions can be found in [Clematis & Gianuzzi 1993][Gregory 1987]. In fact we have realized that all the problems were in fact caused by the unfortunate mixture of problem domains, that is, using conversations to deal with the problems which are not supposed to be handled by conversations and dominate mainly in transactional applications, such as sharing of objects or servers. This issue will be further discussed in Section 2.3.4.

There has been relatively little work on actual implementations of conversations in a distributed system. Implementations of distributed process-oriented conversations are discussed in [Jalote 1986] and [Yang & Kim 1992]. Clematis and Gianuzzi [Clematis & Gianuzzi 1993] addressed issues with structuring conversations in operational/procedure-oriented programming languages such as Ada. Romanovsky and Strigini [Romanovsky & Strigini 1995] offered a limited, but practical method for implementing backward recovery and software diversity within the Ada language. They believe that Ada has sufficient facilities to allow the use of conversations to develop fault-tolerant software and systems. Wellings and Burns [Wellings & Burns 1997] have recently shown by many program examples how Ada 95 can be used to implement atomic actions and achieve software fault tolerance. Forward error recovery is organized in communicating tasks via the simultaneous spreading of exceptions in all tasks involved in an atomic action.

The actual programming of a conversation is another major difficulty associated with the conversation concept. Constructing an application into a sequence of conversations is not a trivial task. The application programmer has to select a boundary composed of a set of checkpoints, acceptance tests and the side walls to prevent information smuggling. This boundary should be integrated well into the structure of processes. [Tyrrell & Holding 1986] suggested a way of identifying adequate boundaries of conversations based on the specification of the application using Petri Nets. [Carpenter & Tyrrell 1991] proposed an alternative solution in which the CSP notation [Hoare 1978] is used to describe the application and conversation boundaries are identified through a trace evaluation, but such traces would cause an explosion of states even for simple applications. In practice, however, it is possible for some special applications to decide on the conversation placement without full trace evaluation [Tyrrell & Carpenter 1992][Tyrrell & Carpenter 1995]. Wu and Fernandez [Wu & Fernandez 1994] proposed a similar method for generating the boundaries of conversations directly from the specification. The specification is described by a high-level modified Petri net, which can be easily transformed into an action-ordered tree. The boundaries are then determined from the tree. Their method is simple and could serve as the basis of a tool to assist in conversation designs.

Finally, there have been a number of recent proposals for "multiparty interaction mechanisms" [Jung & Smolka 1996], intended to coordinate interactions among multiple concurrent processes in distributed programming, e.g. Interacting Processes

(or IP) [Francez & Forman 1996]. However, to the best of our knowledge, none of the proposals published to date provides a means for fault tolerance or for avoiding interference from unrelated processes and parties.

2.3.3 *Object-Based Systems and Atomic (Trans)actions*

Many new architectural developments in the area of distributed computing systems are, to some extent, object-based or object-oriented. Object-oriented techniques, with their modularity, flexibility and reusability features, can be usefully exploited for handling complexity and dependability issues of a distributed system. An object encapsulates some data and provides a set of operations for manipulating the data. An application in an object system can construct its invocations to object operations in the form of transactions.

The transaction concept is an abstraction which allows programmers to group a sequence of operations into a logical execution unit. Traditionally, transactions are expected to satisfy the following four conditions, affectionately known as the ACID properties [Gray 1978][Haerder & Reuter 1983]. *Atomicity*, or the all-or-nothing property, refers to the fact that all the operations enclosed in a transaction must be treated as a single unit; hence, either all the operations are executed, or none. *Consistency* requires a transaction to be correct, i.e. if executed alone, the transaction take the object system from one consistent state to another. *Isolation* means no communication allowed between concurrent transactions. *Durability* implies that the results of a committed transaction is made permanent in spite of failure.

The ACID properties of transactions are usually ensured using two different sets of protocols. Concurrency control protocols ensure execution atomicity, and recovery protocols ensure failure atomicity [Moss 1981]. Execution atomicity refers to the problem of ensuring the overall consistency of the object system, and hence the consistency property of transactions, even when they are executed concurrently. Failure atomicity ensures the all-or-nothing as well as isolation and durability properties.

In fact, the concept of an atomic action has been used in computing science for many years. Dijkstra used them in the specification of semaphores in 1968 [Dijkstra 1968], and Eswaran et al at IBM's San Jose Research Laboratory started to seriously use them in the early 1970s when introducing them, termed transactions, into the database community [Eswaran et al 1976][Gray 1978]. C.T. Davies pioneered the development of the atomic transaction concept [Davies 1973][Davies 1978]. He addressed many concepts concerned with concurrent systems, recovery and integrity within an overall scheme that he called *data processing spheres of control*. Spheres of control are intended to deal with various problems including coordinating multiple processes within recovery regions, sharing partial (uncommitted) data between processes, and controlling concurrency across machine boundaries. However, the descriptions of spheres of control provided little implementation advice for general applications, and

early work on transactions, though influenced by Davies, was much less ambitious in its goals.

Basic transaction systems do not allow for nested transactions or support concurrency within a transaction. Nested transactions [Reed 1978][Moss 1981] extend the flat transaction model by providing the independent failure property for nested transactions, and supporting competitive concurrency within a containing transaction. An in-depth analysis of a large number of algorithms for nested transaction systems can be found in the book by Lynch et al [Lynch et al 1993]. Figure 2.16 shows an example of a transaction that encloses two nested transactions.

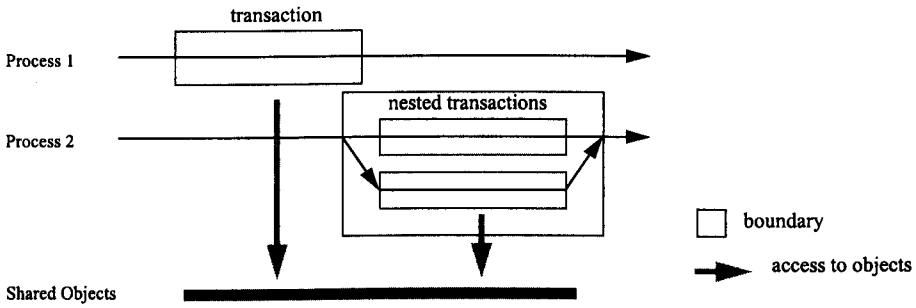


Figure 2.16 Transaction and nested transactions

Traditionally, database systems are modelled as a collection of objects which can only be read or written by transactions. A number of researchers have considered placing more structure on data objects and have shown how this structure can be used to permit more concurrency [Weihl 1989]. In particular, atomic data types are useful for distributed applications, complex design environments, and object-oriented databases. Synchronization algorithms for concurrency control and recovery are designed to exploit the semantics of atomic data types.

In the late 1970s, research begun on distributed databases. One of the important early distributed database systems was R*, developed at IBM's San Jose Lab [Lindsay et al 1984]. Also starting in the late 1970s and early 1980s, a number of research groups began exploring atomic transactions as the basis for structuring distributed systems and applications; out of this work came several languages and systems that successfully combine transaction processing with the object-oriented programming methodology, e.g. Argus at MIT [Liskov 1988], TABS [Spector et al 1985], Camelot, and Avalon at CMU [Eppinger et al 1991][Herlihy & Wing 1987][Spector et al 1985], Clouds at Georgia Tech [Dasgupta et al 1988], Arjuna at Newcastle [Parrington et al 1995], Amadeus/RelaX at Trinity College, Dublin [Taylor et al 1994] and FT-SR reported in [Schlichting & Thomas 1995]. Systems like TABS and Camelot demonstrate the viability of layering a general-purpose transactional facility on top of an operating system. Languages such as Argus and Avalon/C++ go one step further by providing linguistic support for transactions in the context of a general-purpose programming language. Recently, a commercial system using nested transactions has been released called Encina [TransArc 1997], a TP-monitor from TransArc in USA in which most of

the ideas of Camelot have been implemented. In principle application programmers can now use transactions as a unit of encapsulation to structure an application program without having to know how transactions are implemented at the operating system level. However, competitive concurrency is what is assumed to be the major concern here and no explicit support is provided for cooperative concurrency.

There are a great deal of papers and books that have been written on transactions over the last two decades. The book by Bernstein et al [Bernstein & Lewis 1993] provides a good survey of a wide range of concurrent control methods and an informal discussion of correctness issues. The book by Gray and Reuter [Gray & Reuter 1992] gives an in-depth description of many techniques, and is an excellent reference. The book by Lynch et al [Lynch et al 1993] provides a careful rigorous treatment of correctness issues, showing how a wide variety of transaction-processing techniques can be analyzed in a single common framework. Madria [Madria 1997] presented a study on the concurrent control and recovery algorithms in nested transactions and reviewed the most work done in the area of nested transaction modelling. New applications in object-oriented database and mobile environments [Chrysanthis 1993], and in workflow models [Chen & Dayal 1996] were also addressed. Thomasian of IBM T.J. Watson Research Center [Thomasian 1998] has recently given a comprehensive review of concurrency control methods and analyzed their performance in transaction processing.

A number of generalized transaction models have been developed in order to overcome some of the limitations of traditional (flat or nested) transactions, such as lack of support for long-lived actions, cooperatively concurrent activities and multidatabase systems. As a result, it is becoming ever clearer that the traditional transaction model does not provide satisfactory support for cooperation between concurrent activities. Researchers in the area of transactions and databases [Gray & Reuter 1993] are aware of such limitations and problems:

> "The transaction concept has emerged as the key structuring technique for distributed data and distributed computations. Originally developed and applied to database applications, the transaction model is now being used in new application areas ranging from process control to cooperative work. Not surprisingly, these more sophisticated applications require a refined and generalized transaction model. The concept must be made recursive, it must deal with concurrency within a transaction, it must relax the strict isolation among transactions, and it must deal more gracefully with failures."
>
> *Jim Gray* (Foreword for [Elmagarmid 1993])

Most of generalized transaction models and techniques are surveyed comprehensively in the book by Elmagarmid [Elmagarmid 1993]. Typical examples include layered and open transactions [Weikum & Schek 1992], split transactions [Pu et al 1988], multi-colored transactions [Shrivastava & Wheater 1991] and flexible transactions [Warne

1994]. All the new kinds of transactions are more or less based on the relaxation of some of the ACID properties, usually Isolation or Consistency. However, while these sophisticated models lose most of nice properties possessed by the traditional transaction models, they generally provide only limited support for cooperative concurrency. This issue will be further discussed in detail in the next section.

The Venari Project at CMU [Wing 1993] is another attempt of generalizing the traditional transaction model for non-traditional applications. They developed separable modules based on the standard ML modules system to support transactional semantics for different settings, for example, in the absence or presence of concurrency, thereby facilitating different kinds of performance tuning. Concurrency is addressed carefully in two forms: 1) making an individual transaction multi-threaded and 2) allowing multiple transactions to run concurrently (see Figure 2.17). Their work was the first to cast within a programming language (i.e. standard ML) a model of computation that supports multi-threaded transactions [Wing et al 1992]. However, the model they used assumed that there is exactly one thread that enters a transaction and exactly one that leaves a transaction. Such an assumption simply excludes the possibility of regarding their model as a general mechanism for multi-threaded synchronization and coordination which are the essential part of cooperative concurrency.

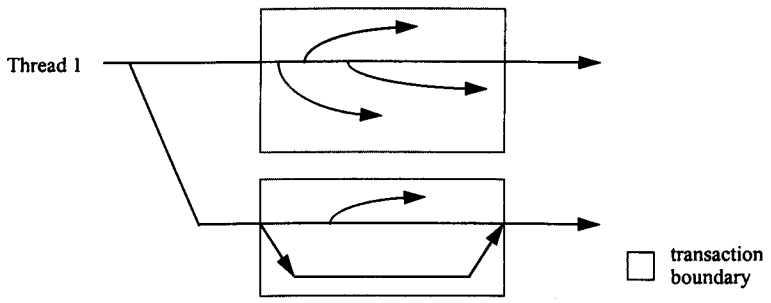


Figure 2.17 Multi-threaded concurrent transactions in Venari

2.3.4 Necessity of an Integrated Framework

It is important to note that there are many dual aspects of conversations (or PM) and atomic transactions (or OM), which are identified carefully in [Shrivastava et al 1993]. The duality leads to a deeper understanding of various fault-tolerant structures and helps the development of new techniques. However, since both models also have many independent characteristics and are developed for different application domains (as discussed previously), conversations and transactions are best viewed as complementary rather than as alternative approaches for a given application — indeed, we would argue that fault-tolerant concurrent software should combine both mechanisms in order to resolve the problems caused by different kinds of concurrency and by hardware and software faults. We will first explain why transaction-based approaches to dealing with cooperative concurrency may have certain limitations, and

then show how an integrated framework can provide more appropriate support for complex applications that involve both competitive and cooperative concurrency.

Generalized Transaction Models

There have been many generalizations of the basic transaction model, most of which provide to some extent support for cooperative activities (e.g. [Skarra 1989][Elmagarmid 1993][Taylor et al 1994]). Generally, these models are based on the concept of nested transactions. At the top-level, a generalized transaction has all the properties of traditional transactions, that is, the ACID properties. However, nested transactions in parallel may be allowed somehow to cooperate. With respect to the permitted degrees of cooperation between nested transactions, there are at least three major methods for extending traditional transactions. The first is to relax strict isolation [Taylor et al 1994]: nested transactions are still serializable but uncommitted results may be shared among them. A nested transaction that uses uncommitted data will depend on the nested transaction that produced the data. Such a nested transaction cannot commit or abort independently and may, once terminated, be required to wait for the commitment of any nested transaction on which it depends before committing. However, since the ACID properties must be retained, only limited cooperation between nested transactions is allowed and the extra performance overhead is further introduced.

The second approach is to enforce the user-defined execution order of nested transactions, which may be specified in the specification of the top-level transaction. (There is also some work concerned with the execution order of both top-level transactions and their nested transactions, for example see [Shrivastava & Wheater 1991].) The atomicity property may be kept for nested transactions but the mechanism for ensuring the correctness conditions defined by the consistency and isolation properties must be extended to permit strong conditions defined by the user-specified execution order. Clearly, such cooperation is restricted by pre-defined execution orders.

By combining the above two methods, a greater degree of cooperation between nested transactions can be achieved. Nodine and Zdonik [Nodine & Zdonik 1984] proposed the substitution of a notion of user-defined correctness for the notion of correctness defined by serializability. Because isolation is not required, correctness conditions on the execution order of operations involved in cooperative nested transactions could be defined for special application purposes [Skarra 1989]. For example, the application programmer can define various correctness conditions based on the relationships like *conflicts*, *patterns* (i.e. specified order), and *triggers* etc.

Problems and Difficulties with Transaction-Based Approaches

Generalized transaction models start from the concept of traditional transactions and suffer inevitably from some original limitations of the traditional model. On the one hand, these models violate the atomicity property of nested transactions, leading to

inconsistency between the top-level *atomic* transactions and nested transactions, and could therefore offer ambiguous semantics of a transaction. Moreover, violation of the atomicity property will complicate built-in recovery mechanisms and impose extra system performance overhead. On the other hand, cooperation among nested transactions, supported by these models, are still restricted since none of them permits true cooperative activities — the boundaries of nested transactions can only be opened up to a limited extent and explicit communications across the boundaries cannot be allowed (or otherwise semantic contradiction on the transaction notion could be caused). In the best case, the application programmer can use the specification mechanism provided by a model to define some operation conditions for a set of related nested transactions and the system then in effect carries out some kind of cooperation by enforcing the specified conditions. However, because no integrated mechanism for possible communication and coordination between nested transactions can be provided to the application programmer, it becomes a particularly difficult task to achieve flexible and fine cooperation.

In contrast, the coordinated atomic (CA) action concept [Xu et al 1995a] is based on a much system-wider view on concurrent, particularly cooperative, activities. A CA action allows different concurrent processes (or threads) to cooperate in performing a joint task by coming together. Explicit communication and coordination among threads are permitted completely but must be enclosed within the boundaries of a CA action. By the use of CA actions, most of the previously-identified limitations in generalized transaction models can be effectively overcome, as we will now address in the following section.

An integrated Solution: The Coordinated Atomic Action Approach

In a concurrent object system, a CA action encloses a joint activity between a group of two or more interacting entities specified by threads. Threads are the agents of computation, and each of them is responsible for executing a sequence of operations on objects. Cooperation between concurrent threads may be based on various different forms of communication and interaction. *Inter-thread cooperation* is modelled in the CA action concept as information transfer via shared objects. Such an abstraction may cover various actual forms of inter-thread cooperation, including inter-thread communication by updating shared objects that have some synchronization mechanisms, or by message passing without requiring shared storage, and of inter-thread synchronization such as condition synchronization (usually no data passed) and exclusion synchronization (usually for shared object schemes).

The CA action approach provides an integrated solution to both cooperative and competitive concurrency. It captures the characteristics of the conversation scheme, or PM, by supporting multi-threaded cooperation within an atomic action structure, and represents the transactional aspects, or the characteristics of OM, by performing a set of operations on a group of objects atomically. Here, threads and transactions are considered as two orthogonal control abstractions which provide different features and

benefits. (In contrast, [Strigini et al 1997] developed an interesting scheme for dividing a heterogeneous system into two subsystems using conversations and transactions respectively.) Depending on one's viewpoint, the CA action scheme can either be seen as an API for nested multi-threaded transactions that imposes appropriate synchronization constraints and allows exception handling, or as a way of extending conversations so as to take advantage of typical transaction mechanisms.

Concurrent transactions have to be atomic. It is not possible to support two-way cooperation and communication between concurrent transactions without breaking down the action boundaries. Most of the generalized transaction models discussed in the last two sections were based on breaking the action boundaries. In contrast, CA actions support multiple threads independent of the transaction abstraction. Performance-related concerns are another reason of seeking for an integrated approach. Transactions require runtime mechanism to support protocols for locking, logging, committing/aborting, and crash recovery etc. There are practical cases where cooperative concurrency is desired without the performance overhead imposed by transactions. Even if the application programmers were to write an example shown in Figure 2.18 with transactions, they probably would not want to incur the cost of making each isolated part (e.g. *t1*, *t2*, *t3*, *t4*, and *t5* in the figure) an individual transaction. In this aspect, CA actions provide flexibility and performance benefits.

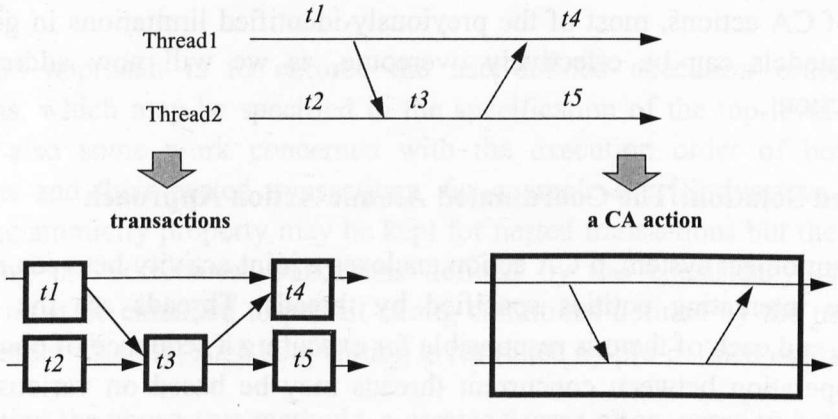


Figure 2.18 Two-way communication: transactions versus CA actions

Most models for fault-tolerant distributed systems take only hardware-related faults into account. That is, if a software system is known to be in an error-free state upon entry to an action, the action will be terminated normally (committed), producing the intended results, or aborted (due to hardware failures or conflict access), producing no results. However, residual software design faults in the code of an action could be a major cause of some erroneous, but committed results (see the detailed discussion in Sections 2.1 and 2.2.) CA actions are intended to provide a basis for coping with both hardware-related failures and software design faults. The CA action framework allows the controlled usage of both backward and forward error recovery techniques (e.g. involving compensatory messages to external activities, i.e. environment of the system, that may have been affected by erroneous output from the system). This could be very

valuable for systems that interact with environmental objects that cannot be simply backed up. In particular, software fault tolerance properties possessed by CA actions will be further investigated in Chapter Four.

2.4 System-Level Support and Environments

In order to make software fault tolerance effective and feasible on a routine basis, one of the important problems that has to be solved is the development of appropriate linguistic support and easy-to-use environments, which should not complicate greatly the program's implementation, readability, and maintenance.

2.4.1 Typical System Examples

Early environments for supporting fault-tolerant software have focused on one or other of the two classical approaches: recovery blocks and *N*-version programming. Two systems that support the development of recovery blocks were reported in [Anderson & Kerr 1976][Shrivastava 1979] respectively (see some details in Section 2.2). The DEDIX system [Avizienis et al 1985][Avizienis et al 1988a] was actually a supervisor program for research use with *N*-version programming, and was implemented as an application-level Unix package. DEDIX provides support for concurrent execution of different versions and voting on results, but it is not quite suitable for multi-version design at the module level and inter-process concurrency.

Ancona et al [Ancona et al 1990] developed a system architecture for fault tolerance in concurrent software and described a mechanism, called the *Recovery Metaprogram* (RMP), for the incorporation of fault tolerance functions into application programs. They give application programmers a single environment that lets them selectively use appropriate fault tolerance schemes, including recovery blocks, *N*-version programming, programmer-transparent coordination, and conversations.

The proposed architecture contains three basic components: the application program, the RMP and the kernel. The application programmer must define the software variants and the validation test, and indicate which portions of the application program are involved in fault tolerance. The RMP implements the controllers and the supporting mechanisms for four different schemes, inserting a number of breakpoints in the program. When a breakpoint is reached, the application program is suspended and the kernel activates the RMP which takes actions to support the fault tolerance scheme chosen. The RMP is then suspended, and the application program is reactivated until the next breakpoint is reached. Figure 2.19 shows the control flow between an application program and the recovery metaprogram. Ozaki et al. [Ozaki et al 1988] described a possible implementation of some primitives required by the RMP. The implementation of the RMP approach however incurs an additional cost in the form of intensive context switches and kernel calls.

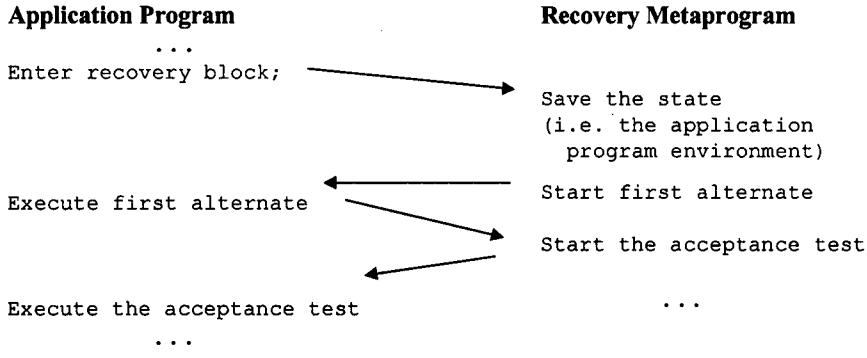


Figure 2.19 Control transfer between an application program and the RMP

Purtilo and Jalote [Purtilo & Jalote 1991] introduced an environment that supports execution of programs using both recovery blocks and *N*-version programming in a uniform manner. Their environment allows module-level software redundancy, versions to be written in different programming languages, and executed on different machines. The environment has been developed for use on Unix-based hosts and currently runs on a network of Sun and DEC workstations. Figure 2.20 gives an example of the module definition for recovery blocks. This example shows how the system supports use of the distributed recovery block scheme. Three versions, whose object files are `c_string.o`, `p_string.o`, and `l_lisp.l`, are written in C, Pascal and Lisp and executed on respective machines named `brillig`, `slithy`, and `tove`. But the system made no provision for concurrent programming, and did not address the issues with inter-process cooperation and distributed error recovery.

```
module proc {
  RB module
  interface: proc(proc_list)
  implementation: "c_string.o@brillig"
  implementation: "p_string.o@slithy"
  implementation: "polylisp@tove l_string.l"
  acceptance: "at.o"
  error-handler: "handler.o"
}
```

Figure 2.20 An example of the module definition for RB

Tso and Shokri of SoHaR Incorporated reported a testbed environment for the construction and evaluation of software fault tolerance systems, called ReSoFT [Tso & Shokri 1996]. The ReSoFT environment comprises a library of reusable components implemented in Ada 95 from which a variety of fault-tolerant software systems can be built. The schemes supported by the environment include recovery blocks, *N*-version programming, re-try blocks, and *N*-copy programming. One of the new characteristics of ReSoFT is a set of graphical tools to facilitate the construction of fault-tolerant software, monitoring, and testing of the software through fault-injection. However, this testbed still provides no support for concurrency.

There are also a few papers that deal with notation issues. [Liu 1992] proposed a design notation for a wide class of fault-tolerant software structures, mainly offering generality and flexibility in a modular fashion. [Bondavalli & Simoncini 1992] showed that their BSM design description language is sufficient for expressing the typical structures of software fault tolerance, such as recovery blocks and *N*-version programming, without requiring semantic extensions. [Silva et al 1996] developed a simplified programming model for communicating process systems and proposed a notion of RP-actions (or actions for resilient processes). Figure 2.21 illustrates an example of a single process structured as a set of RP-Actions. However, because the RP-action scheme supports inter-action communications in a way similar to transactional settings, for certain applications the overhead mainly contributed by checkpointing can be as high as 606% [Silva et al 1996].

```
...
while(true) do {
  begin_action(error_code,...);
  if (time-out)
        <emergency block>;
  if (first_execution or transient_fault)
        <primary block>
        if (assertion)
          abort_action(error_code);
  if (software_fault)
        <alternate block>
        if (assertion)
          abort_action(error_code);
  if (permanent_fault)
        <entering a fail-safe state>
  if (acceptance_test)
        abort_action(error_code);
}
```

Figure 2.21 An example of a single process structured as a set of RP-actions

2.4.2 Reusable Components Supporting Software Fault Tolerance

Strigini and Avizienis [Strigini & Avizienis 1985] were the first to suggest the use of a reusable toolset to develop fault-tolerant software. They argued that such a toolset should include 1) language-level support for exception handling, 2) low-level run-time support for message passing, state saving and restoring, and application-independent adjudicating, 3) libraries for recovery blocks, atomic actions etc., and 4) system configuration tools. Huang and Kintala of AT&T Bell Labs. [Huang & Kintala 1995] developed three software reusable components in C that provide software fault tolerance in the application layer, supporting fault-tolerant structures like checkpointing and recovery, replication, recovery blocks, *N*-version programming, exception handling, re-try blocks etc. Their modules have been ported to a number of UNIX platforms, already applied to some new telecommunications products in AT&T

and the performance overhead due to these components has been shown to be acceptable. But these components provide no support for inter-process communication and cooperation.

The dependability research group at Newcastle has recently developed a set of reusable components in C++, providing high-level object-oriented programming interfaces for software fault tolerance. The recent extension of C++ to include generic classes and functions ("templates"), and exception handling ("catch" and "throw") makes it possible to implement both forward and backward error recovery in C++ in the form of reusable components that separate the functionality of the application from its fault tolerance [Rubira & Stroud 1994]. Figure 2.22 shows an example of using the "catch" and "throw" structure to implement error recovery. More generally, such facilities show prospect of providing a convenient means of achieving high levels of reuse. This would apply both to general software components implementing various fault tolerance strategies (including generalizations and combinations of recovery blocks, and *N*-version programs, and encompassing the use of parallelism) and to application-specific software components [Xu et al 1995c].

```
class SafeCollection : public FastCollection
{     public:
            virtual boolean find(int);
            virtual int min();
}
//below a safe implementation of the min function
int SafeCollection :: min()
{     try
      {
            return FastCollection ::min();
      }
      catch(...)
      {
            return SimpleCollection::min();
      }
}
```

Figure 2.22 An example of implementing error recovery in C++

However, there remain certain strategies and types of structuring that cannot be implemented entirely (or at any rate elegantly) in a language like C++ even given such mechanisms as generic functions and inheritance. Instead, the programmer who wishes to employ these strategies has to obey certain conventions. For example, the application programmer who wishes to make use of the reusable C++ classes would have to include explicit calls in each operation of an object to facilities related to the provision of state restoration.

Adherence to such conventions can be automated, by embodying them into a somewhat enhanced version of C++ and using a pre-processor to generate conventional C++ programs automatically. Although the pre-processor approach can be quite practical it

does have disadvantages. In particular the language provided to application programmers becomes non-standard since programmers have in some circumstances during program development to work in terms of the program generated by the pre-processor, rather than of the one that they had written. The alternative, that of leaving it to the programmer to adhere to the conventions, is of course a fruitful source of residual program faults. But developing a new language that provides adequate syntax and runtime support to enable the implementation of various software fault tolerance schemes could cut the work off from the mainstream of programming language developments and thus have difficulty in achieving wide acceptance.

2.4.3 *Reflective System Architecture*

It is important to notice a fundamental difference between simple replication and multi-version design; the former could be made transparent completely to the application programmer and performed automatically by an underlying support mechanism, but the latter requires to certain extent direct effort from the application programmer. Simple (thus easy to check) language features with powerful expressibility are neverhteless particularly helpful in properly specifying software variants and the adjudicator. The further issue is how the underlying support mechanisms can be provided in a more natural and modular manner rather than by an *ad-hoc* method such as system calls. Recent developments in the object-oriented language world, under the term "*reflection*" [Maes 1987], show considerable promise in this regard.

A reflective system can reason about, and manipulate, a representation of its own behaviour. This representation is called the system's meta level [Agha et al 1992]. Reflection improves the effectiveness of the object level (or base level) computation by dynamically modifying the internal organization (actually the meta level representation) of the system so as to provide powerful expressibility. Therefore, in a reflective programming language a set of simple, well-defined language features could be used to define much more complex, dynamically changeable constructs and functionalities. As for the development of fault-tolerant software, it could enable the dynamic change and extension of the semantics of those programming features that support software fault tolerance concepts, whereas the application level (or object level) program is kept simple and elegant [Xu et al 1995c]. Although the C++ language itself does not provide a meta level interface, Chiba and Masuda [Chiba & Masuda 1993] describes an extension of the C++ language to provide a limited form of computational reflection, called Open C++. Figure 2.23 illustrates how base level method invocation can be trapped and be then adjusted at the meta level in Open C++. Some experimental results in [Xu et al 1996] have shown that the run-time overhead introduced by a reflective operation call in Open C++ is insignificant in comparison with other overheads imposed by a software fault tolerance scheme.

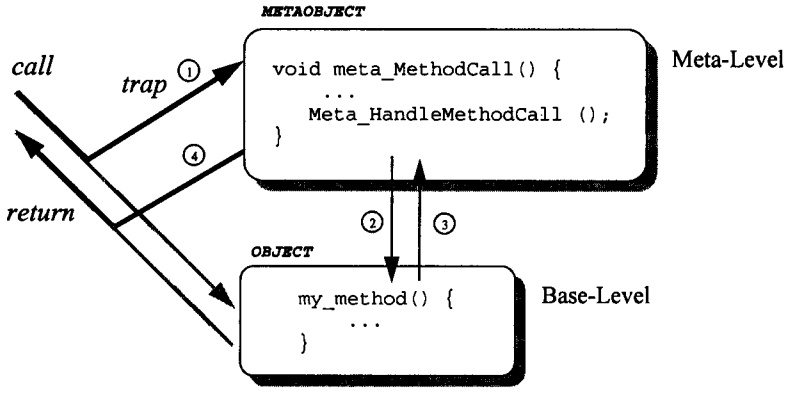


Figure 2.23 Invocation trapping in Open C++

Collaborative work between LAAS-CNRS and Newcastle has developed several case studies and prototypes using Open C++ to implement fault-tolerant applications [Fabre et al 1995]. The use of metaobject protocols to implement atomic objects in transaction-based systems is presented in [Stroud & Wu 1995]. But none of them have addressed the issues of software fault tolerance. [Fabre & Pérennou 1998] reported the recent development at LAAS-CNRS. They presented a reflective system in Open C++ called FRIENDS which provides libraries of metaobjects for hardware fault tolerance, secure communications and group-based distributed applications. The authors also gave an excellent summary of the advantages and drawbacks of a metaobject approach for building fault-tolerant systems. Recently, Beder and Rubira [Beder & Rubira 1998] reported some progress in their efforts in the definition of a reflective framework for developing dependable software based on patterns and metapatterns [Buschmann et al 1996].

When considering support for software fault tolerance in concurrent object-oriented programming, we face a greater challenge because, though a large number of different models for concurrent object-oriented programming have been proposed, none has yet received widespread acceptance. There exist only a few tentative proposals for treating concurrent error recovery such as the Arche language [Issarny 1993a]. However, the reflection technique seems to be a more promising approach to the structuring of concurrent object-oriented programs [Yonezawa & Watanabe 1989].

2.5 Summary

In this chapter we have provided a comprehensive overview of the techniques for building fault-tolerant software, covering both sequential and concurrent programs as well as environments that support the development of fault-tolerant software. Although various other types of technique for sequential programs were outlined here, such as data diversity and environment diversity, we have focused our attention on techniques that are essentially based on diverse designs, i.e. software redundancy through functionally equivalent software components. One of the major problems associated with such approaches is that all software versions may suffer from a common-mode

failure. This would prevent distinction between incorrect and correct outputs, thereby causing a possible system failure. Nevertheless, the analytic results based on some common experimental data have shown that multi-version software can offer additional dependability over and above what we can gain any other way, since we are not able to actually make a single-version system extremely dependable with the current state of the art. When the given cost allows the production of multiple versions with very high quality, multi-version software will be perhaps the unique way of achieving ultra-high dependability for critical applications. However, when the time and budget are limited, it is unclear whether multi-version software is still superior to a single-version system. This is because for the same cost each version of the multi-version software may have much less quality than a single-version system. The development of advanced fault-tolerant software techniques with improved dependability figures will certainly help a software designer to make a correct decision as to which technique, multiple versions or a single version, is likely to be most effective for a given application with limited budget.

Fault-tolerant software techniques for concurrent and distributed programs are mainly concerned with system structuring and providing different ways of controlling and confining information flows among concurrent activities. We have examined two well-known system models used in different application domains, i.e. the process-conversation model and the object-action model, and the related schemes and techniques. For a more general and thus more complex application, we have argued that an integrated model is needed that uses processes (or threads) and atomic actions as two orthogonal control abstractions so as to facilitate the control of both cooperative and competitive concurrency. There are two similar techniques that are based on the combined model, the Venari multi-threaded transaction scheme and the coordinated atomic action scheme. However, to be a practical fault-tolerant software technique, both schemes require further investigation and development, especially in exception handling and error recovery aspects.

Existing experience with the environments that support the development and execution of fault-tolerant software has been limited, and most examples provide no support for concurrency. Reusable components in the form of libraries and metaobject-based approaches with reflective capabilities have been quite successful, including such examples as the reusable components used in the AT&T communications products and the FRIENDS system developed in LAAS-CNRS. However, both systems are designed mainly to handle hardware-related faults. Quite what reflective capabilities are needed for what forms of software fault tolerance, and to what extent these capabilities can be provided in object-based or object-oriented programming languages, and allied to the other structuring techniques such as atomic actions, remain to be determined. In particular, the problems of the combined provision of significant software fault tolerance and hardware fault tolerance, and of evaluating cost-effectiveness, are likely to require much further effort.

Paginated blank pages are scanned as found in original thesis

No information is missing

Chapter 3

Advanced Schemes for Designing Fault-Tolerant Software

This chapter introduces two advanced schemes, i.e. $t/(n-1)$-variant programming ($t/(n-1)$-VP) and self-configuring optimal programming (SCOP), addressing the issues of improving software reliability and achieving the dynamic trade-off between dependability and efficiency, respectively. Dependability and efficiency improvements obtained by these schemes are modelled and evaluated using a Markov approach, and the analytic conclusions are also supported by an empirical comparison.

3.1 $t/(n-1)$-Variant Programming

As discussed in Chapter Two, software fault tolerance usually requires the application of design diversity in which two or more variants of a component for redundant computations are independently designed to meet a common service specification. Variants are aimed at delivering the same service, but implemented in different ways in the hope that they do not contain the same design faults. Since at least two variants are involved, tolerance to design faults necessitates an adjudicator (i.e. a decision algorithm) that determines a single (assumed to be) error-free result based on the results produced by multiple variants.

It must be recognized that the success of a fault tolerance scheme depends to a great extent upon its adjudicator and unreliability in the adjudicator can have a dramatic impact on the overall system reliability [Anderson & Lee 1981]. The design for a highly reliable adjudicator generally requires that

1) the adjudication mechanism and variants being checked are as independent as possible, so that they are very unlikely to be affected by common faults or related faults;

2) the mechanism itself must be simple enough to provide a good guarantee of its reliability and the system performance.

The traditional mechanisms are not entirely satisfactory. In the recovery block software, an acceptance test is used in its adjudication mechanism to provide a last line of detecting errors, but since the test is system-specific, and as such very little specific guidance can be given for its construction, it is difficult to ensure that the acceptance test and variants will be independent of each other. To overcome this problem, many schemes adopt an adjudication mechanism that simply selects the results by comparing the outputs of multiple variants. However, a practical adjudicator used in NVP is much more sophisticated than the early idea of a simple majority vote, while adjudication mechanisms constructed in NSCP are too simple to effectively detect the related faults that may occur in the active self-checking components. We develop an alternative here, called *t/(n–1)-Variant Programming* (*t/(n–1)*-VP), which exploits several new research results in the area of system fault diagnosis [Barborak et al 1993] for the design of a simplified adjudication mechanism. Our proposed scheme has several favourable characteristics, including:

1) it has a potential ability to tolerate multiple related faults among variants,

2) the adjudication mechanism is simple and requires only $O(n)$ result comparison steps,

3) correct service can be delivered even when the number of faulty variants exceeds the bound t in some fault situations, and

4) there are possible forms of graceful degradation.

3.1.1 Description of the t/(n–1)-VP Scheme and an Example

In the theory of system-level fault diagnosis (see [Barborak et al 1993] where further references can be found), a particular diagnosability measure, denoted as $t/(n-1)$-diagnosability, was first introduced in [Friedman 1975]. Its diagnosis goal is, for a system composed of n units, to isolate the faulty units to a set of size at most $(n-1)$, under the condition that the number of faulty units is at most t. That is, at least one unit exists such that it is not in the set of size $(n-1)$ and can thus be unambiguously identified as fault-free, provided that the system itself is $t/(n-1)$-fault diagnosable and the number of faulty units in the system does not exceed the bound t. The $t/(n-1)$-diagnosis technique can be therefore used to select a single correct result from the results generated by n replicated software modules (of independent design).

We can benefit from the utilization of $t/(n-1)$-diagnosis since this special diagnosis measure cuts down significantly the requirement on the number of tests (i.e. the number of result comparisons) relative to previous diagnosis schemes. It is thus possible to use the idea behind the $t/(n-1)$-diagnosis technique to construct a simple, but dependable adjudication mechanism. Based on some theoretical results of $t/(n-1)$-diagnosis (see a subsequent discussion and Appendix *A*), we develop a new scheme for tolerating hardware and/or software faults. Our description of this scheme is first in terms of

application to software fault tolerance, but the approach can also be implemented with hardware.

A general $t/(n-1)$-VP architecture can identify the correct result from a subset of the results of n software modules (or variants), provided that the number of faulty modules in the architecture does not exceed t (i.e. it can tolerate at least t software faults). A *syndrome* is defined here as a set of information used by an adjudicator or a diagnosis algorithm to perform its judgement as to the correctness of a result, in general including those results produced by variants. The semantics of $t/(n-1)$-VP can be expressed more directly as follows:

1) each of n independently designed software variants is executed in parallel;

2) just some of their results are compared to produce a syndrome;

3) using the syndrome, a diagnosis program performs $t/(n-1)$-diagnosis and attempts to select a presumably correct result as the system output (i.e. through switching of the results); if no acceptable result is identified, the system will invoke spare software variants, if any exist, or simply signal an exception.

We now use a concrete example to demonstrate the ability of $t/(n-1)$-VP to tolerate software faults, and then address its effectiveness for any given n and t. Two classes of software faults are distinguished: independent faults and related faults [Laprie et al 1990][Arlat et al 1990]. Independent faults occur in single variants or in the adjudication mechanism, while related faults can take place among multiple variants and among the adjudicator and one or more variants. Figure 3.1 shows a $t/(n-1)$-VP architecture where $n = 5$ and $t = 2$.

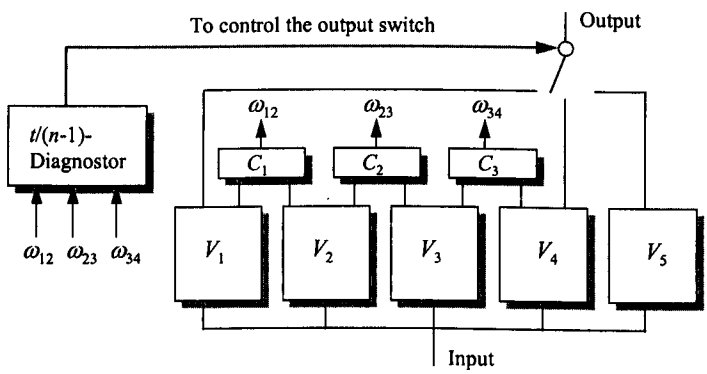


Figure 3.1 A $t/(n-1)$-VP architecture with $n = 5$ and $t = 2$

This 2/(5–1)-VP architecture consists of five independently designed software modules, called variants V_1, V_2, ..., and V_5, which are executed in parallel in a framework that is intended to cater for up to two simultaneous software faults. Three comparators C_1, C_2, and C_3 are placed at the outputs of variants V_1, V_2, V_3, and V_4 to perform error detection, where C_i compares the results of V_i and V_{i+1} (i = 1, 2, 3) and generates the

test outcome $\omega_{i,i+1}$. Three (comparison) test outcomes ω_{12}, ω_{23} and ω_{34} constitute a syndrome. In particular, the test outcome $\omega_{ij} = 0$ (1) if the results of the variants V_i and V_j agree (disagree). A diagnosis program, the $t/(n-1)$-diagnostor, selects one of the results of V_1, V_4, and V_5 according to the value of the syndrome, and switches service delivery, i.e. the system output, to the selected result. The adjudicator of the architecture is implemented by the three comparators, the $t/(n-1)$-diagnostor and the output switch. Note that V_2 and V_3 are not connected to the output switch and V_5 is not connected to a comparator. However, this architecture is $t/(n-1)$-diagnosable for $t = 2$: the diagnostor can always select a correct result *provided* that the number of, independent or related, faults in variants does not exceed two.

Let r_1, r_2, ..., and r_5 be the results of variants V_1, V_2, ..., and V_5 respectively. Table 3.1 gives all possible syndromes and the corresponding results that can be unambiguously diagnosed as correct while assuming that no more than two faults occur simultaneously. For example, in the case that $\omega_{12} = 0$, $\omega_{23} = 1$ and $\omega_{34} = 0$, a single correct result cannot be simply identified from among those produced by variants V_1, V_2, V_3, and V_4. We can however infer from the syndrome that two or more of the variants V_1, V_2, V_3 and V_4 have generated incorrect results because one single fault cannot lead to such a syndrome. Hence the result of V_5 must be correct. In the case where $\omega_{12} = \omega_{23} = \omega_{34} = 0$, either all of the variants V_1, V_2, V_3 and V_4 have to be correct or all of them have to be incorrect. By the previous assumption that $t = 2$, these results should be classified as acceptable. Following a similar method, we can analyze other cases to determine the correct results. In fact, Table 3.1 may be viewed as a simple diagnosis algorithm for the specific architecture. In this table, it should be noted that at least one of results r_1, r_4 and r_5 must be correct for a given syndrome. Accordingly, this architecture can deliver the correct system output by choosing just among the results of three variants V_1, V_4 and V_5.

ω_{12}	ω_{23}	ω_{34}	Presumably Correct Results
0	0	0	r_1 r_2 r_3 r_4
0	0	1	r_1 r_2 r_3
0	1	0	r_5
0	1	1	r_1 r_2
1	0	0	r_2 r_3 r_4
1	0	1	r_5
1	1	0	r_3 r_4
1	1	1	r_5

Table 3.1 Possible syndromes and result selections for the 2/(4)-VP example

3.1.2 General t/(n–1)-VP Architecture

Figure 3.2 shows a Petri-net execution model for a general $t/(n-1)$-VP scheme. In the interests of simplicity and brevity, we omit some mechanisms in the figure such as input data consistency. Note that the label K associated with an arc means that the ensuring transition may be fired only after K tokens have accumulated in the preceding place. In particular, K can be less than n for a given diagnostic algorithm; see the architecture shown in Figure 3.1 for example.

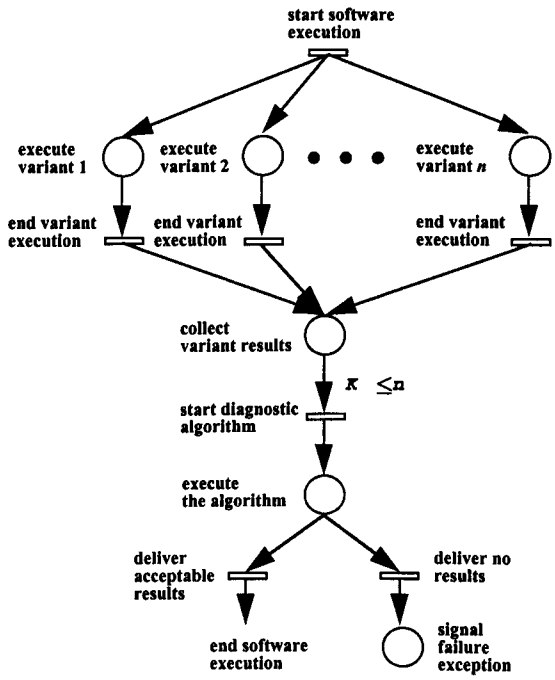


Figure 3.2 Execution model of the $t/(n-1)$-VP scheme

Unlike the NVP scheme and its variations, $t/(n-1)$-VP does not have to make pairwise comparisons among the results of n variants in order to identify a presumably correct result. It is however interesting to study how many result comparisons (corresponding to the comparators illustrated in Figure 3.1) are normally required for a general $t/(n-1)$-architecture. In the simplest case that $n = 3$ and $t = 1$, one comparator is necessary and sufficient for $t/(n-1)$-diagnosis — the third result must be acceptable when the two compared results disagree; otherwise they can be identified as correct.

Larger n and t require a more deliberate comparison assignment among the results of multiple variants so as to guarantee $t/(n-1)$-diagnosability. For example, result comparisons of n variants could be organized into a form of chains, where the comparator C_i ($1 \le i \le n-1$) compares the results of variants V_i and V_{i+1}. Alternatively, result comparisons may be organized into a more complex structure, called $H_{2r,n}$, such that the result of variant V_i ($1 \le i \le n$) is compared with that of V_j if and only if $i-r \le j \le i+r$ (mod $n+1$, $r = 1, 2, 3, ...$). Theorem 3.1 shows several sufficient conditions on the

result comparison assignments of the systems that are $t/(n-1)$-diagnosable; its proof is given in Appendix *A*.

> *Theorem 3.1:* A system S composed of n units (or software variants) is $t/(n-1)$-diagnosable if $n \geq 2t + 1$ and the assignment of result comparisons in the system S contains at least
>
> 1) a chain of $2t$ units for $1 \leq t \leq 2$;
>
> 2) a chain of $2t + 1$ units for $3 \leq t \leq 4$;
>
> 3) an $H_{2r,n}$ structure with $r = 1$ for $5 \leq t \leq 6$;
>
> 4) an $H_{2r,n}$ structure with $r \geq (t-1)/5$ for $7 \leq t$.

Because the major aim of this section is to show how the $t/(n-1)$-diagnosis technique could be applied to the design of software fault tolerance schemes, we will not further discuss this particular technique itself. The diagnosis algorithms with respect to the testing assignments in the above theorem have been developed in [Xu 1991] for chains and in [Xu & Huang 1995] for $H_{2r,n}$-type systems. For practical values of n (e.g. $3 \leq n \leq 10$), a $t/(n-1)$-VP architecture uses only $O(n)$ comparators and contains a simple diagnosis algorithm with linear complexity. The adjudicator in such an architecture would be simpler than a voter used in NVP (which has to be based on $O(n^2)$ result comparison steps).

It is important to realize that for any fault tolerance scheme the correctness of results output by the system cannot always be guaranteed (e.g. when more than t faults have occurred). Moreover such fault situations cannot be detected completely. However this does not present severe problems; there are acceptable probabilities of catastrophic events in practice (e.g. an aircraft computer system is usually acceptable if the probability of failure is less than 10^{-9} per hour in a ten hour flight). Dependability studies that have been performed for practical fault-tolerant systems can be used to determine the probability of the occurrence of t faults. This helps to make an appropriate design decision as to which scheme is likely to be most effective and how many variants are sufficient for a particular application. Additional fault detection and exception handling techniques [Cristian 1995] can also be used to improve fault coverage and fault tolerance. In the 2/(5–1)-architecture of Figure 3.1, for example, exception-handlers can be incorporated into the variants. The function of the handlers in a variant is to handle any errors that are detected during the execution of the variant, signalling an exception to the diagnostor if necessary. The diagnostor comes to final decision according to the value of the syndrome and the exception signals received so far: either it delivers a presumably correct result, or signals a failure exception.

3.1.3 Comparison with Other Schemes

The $t/(n-1)$-VP scheme has some resemblance to other fault tolerance techniques that have been previously proposed and examined, especially with those requiring the use of result comparisons such as NVP and NSCP. The fact that the variants are executed in parallel necessitates an input consistency mechanism and a synchronization regime, based essentially on wait and send primitives, and incorporating a time-out mechanism. However, in each case there are significant and fundamental distinctions. Correct results in $t/(n-1)$-VP are not obtained by majority vote (as in NVP), or by detecting and discarding erroneous results (as in NSCP), but by $t/(n-1)$-diagnosis.

NVP

It could be argued that $t/(n-1)$-VP is only a variation of NVP; however, in our opinion, the majority voting check is an integral part of NVP, and each of N software versions in NVP is of equal importance. In marked contrast, the $t/(n-1)$-VP scheme does not try to find a majority of n results, but just to identify a presumably correct result. It can therefore deliver correct results with some probability even when the majority of results of n variants are incorrect. Moreover, $t/(n-1)$-VP has more flexible architectural features. In the architecture of Figure 3.1, the variant V_1 can be considered as being active, actually delivering the system output in the absence of faults; the variant V_4 and V_5 are used as "hot" spares, and V_2 and V_4 are only exploited for detecting errors and producing test outcomes. In addition, NVP requires that all variants should be designed to produce the results that are essentially identical. This constraint can be loosened in the $t/(n-1)$-VP approach. While the primary variant V_1 in the 1/(5–1)-VP architecture should attempt to produce the desired output, the spare variant V_5 may only attempt to provide a degraded service. In this form, the $t/(n-1)$-VP architecture can be used to implement a type of graceful degradation.

NVS

In principle, $t/(n-1)$-VP is also different from NVS (a form of sequential NVP [Grnarov et al 1980].) The $t/(n-1)$-VP method is based on so-called hot-standby redundancy, whereas NVS utilizes the cold-standby technique. More precisely, in the case that the results of the first two variants disagree (assuming $N = n = 3$), $t/(n-1)$-VP will select the result of the third variant, which has been available, as the system output through the result switch. NVS however has to first execute the third variant on the same set of input values and then make a further decision by searching for a majority of the results. This validation process requires extra execution time for the third variant and for the final decision. Clearly, in comparison with our scheme, NVS has relatively poor predictability of task completion time and may be inappropriate for certain time-critical applications.

NSCP

It could be argued that $t/(n-1)$-VP is somewhat similar to NSCP. However, a fundamental distinction between the two schemes concerns their capacity for tolerating related faults. NSCP will fail (and perhaps cause catastrophic consequences) whenever the two variants that form the active self-checking component produce identical, but incorrect results (no matter how many spares are still available). In contrast, the $t/(n-1)$-VP scheme can tolerate up to t (independent or related) faults; that is, it can deliver correct service even if t faulty variants compute identical incorrect results.

RB

Finally, from the previous overview of software fault tolerance schemes, it is evident that the $t/(n-1)$-VP approach is quite distinct from the recovery block concept. Like NVP and its variations, $t/(n-1)$-VP is complementary in many respects to RB. Recovery blocks can be more appropriate for those systems where hardware resources are limited and comparison-based adjudicators are inappropriate (a discussion of the relative advantages and disadvantages of NVP and RB has been given in Chapter Two). In the interests of simplicity and brevity, we will focus in the evaluation part (i.e. Section 3.3) on the comparison of $t/(n-1)$-VP with NVP and NSCP without further discussing the recovery block approach.

3.2 Self-Configuring Optimal Programming

Most software fault tolerance methods that address dependability issues solely are often inefficient (this will be discussed in detail in Section 3.2.1). If a software system has to be structured to treat software faults, efficiency will certainly remain an important aspect of its quality. Possible resolutions of the efficiency problem are the subject of this section.

Since the kind of applications which require software fault tolerance are often also likely to have stringent efficiency requirements, a good use of the available resources, both in space (hardware) and time (repetition), is highly desirable. In fact, there has been growing research interest in combining both aspects, typical efforts including Tai's performability-driven adaptive fault tolerance [Tai 1994] and Stankovic and Ramamritham's reflective architecture for real-time OS's [Stankovic & Ramamritham 1995].

3.2.1 Software Fault Tolerance versus Software Efficiency

Let us briefly revisit several typical schemes for software fault tolerance and examine their efficiency aspects. Software variants are organized in recovery blocks in a manner similar to the standby sparing techniques (dynamic redundancy) used in hardware and may be executed serially on a single processor. The execution time of a recovery block

is normally that of the first variant, acceptance test, and the operations required to establish and discard a checkpoint. This will not impose a high run-time overhead unless an error is detected and backward error recovery is required. In this regard, RB is highly efficient. Limitations of the RB method are mainly connected with the acceptance test, which is usually derived from the semantics of a given application. Such an acceptance test will introduce a run-time overhead which could be unacceptable if the test is complex, but the development of simple, effective acceptance tests is a difficult task.

NVP, NSCP and $t/(n-1)$-VP avoid using an acceptance test by taking advantage of parallel execution of multiple variants and result comparisons (although sequential execution is conceptually possible just as parallel execution of RB alternates can be performed in a distributed computing system [Kim & Welch 1989]). The adjudication mechanisms used in these three schemes are usually based on result comparison and thus independent of semantics of the applications. The probability of common mode failure between the adjudicator and the variants is relatively low in these schemes. When variants are executed in parallel, NVP, NSCP and $t/(n-1)$-VP have relatively fixed response time (without repetition), thereby guaranteeing timely responses even in the presence of faults. However, it is important to notice that these architectures utilize redundancy in a static manner and always execute all of their variants regardless of the (normal or abnormal) state of the system. They are intended to tolerate the maximum number of faulty components that may be present in the system; but, since such a "worst case" rarely happens, the amount of resources consumed is often higher than necessary. In this sense, they are not efficient.

All the fault-tolerant approaches require some extra space or extra time, or both. Figure 3.3 summarises space-time overheads in software fault tolerance schemes. Space is defined as the amount of hardware (e.g. the number of processors in a distributed system) needed to support parallel execution of multiple variants. Time is viewed here as the physical time needed to execute one or more variants sequentially. It is important to notice that efficient or optimal use of the available resources generally requires dynamic management and conditional execution of the available software variants. This, as is shown in the diagram, should come with a dynamic trade-off between full parallel execution and totally sequential execution of variants. Unfortunately, no existing schemes attempt to provide such a dynamic space-time trade-off though it can in fact be achieved and so we would argue it should be provided. As a possible solution, we propose in the next section a new scheme, called Self-Configuring Optimal Programming (SCOP), which improves the efficiency of software fault tolerance by diminishing the possible waste of resources, without compromising software dependability.

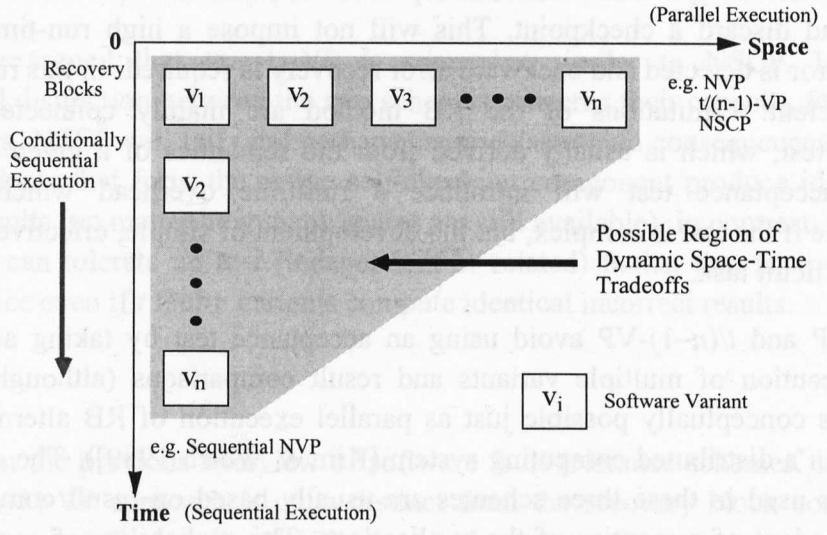


Figure 3.3 Space and time redundancy in fault-tolerant software

In order to use redundancy in a dynamic or conditional manner, a scheme has to decide, at appropriate intermediate points of its execution, which of the following three execution states has been reached: i) *End-state E*, that is, there exists a result that meets the stated *delivery condition* and can thus be delivered; ii) *Go(ing on)-state G*, that is, there is no result that meets the condition, but it is still possible to obtain such a result if further (space or time) redundancy is employed; or iii) *Failure state F*, that is, there is no further possibility of producing a result that meets the condition. In consideration of efficiency, our scheme, SCOP, structures its execution in sequential phases, configuring its variants in a dynamic and adaptive fashion. Each phase includes the execution of the least number of variants, i.e. one which will lead to an end-state if no error is detected. At the end of a phase, if a result that meets the delivery condition is found, SCOP simply outputs it and stops any further execution.

In reality, conditions for delivering a result are usually embedded in the adjudicator explicitly or implicitly. Some examples for canonical delivery conditions are:

1) a result is deliverable if its probability of being correct is assessed to be equal to or higher than a previously determined bound α; or

2) the results are passed on if they constitute the majority and the erroneous results (i.e. the minority) are eliminated.

Note that different conditions generally have different fault coverages though some conditions seem to be very similar. In further consideration of efficient use of the available resources, SCOP is devised to admit different delivery conditions, thus becoming parametric with respect to the level of fault tolerance.

Dynamic redundancy for the purpose of space-time trade-off is a classical idea, e.g. Duplicated Configuration with a Spare and NMR with Spares used in hardware. Similar schemes have been applied to redundancy management in multiprocessors [Lombardi 1985] or in distributed computing [Babaoglu 1987]. However, all these schemes take hardware faults into account only, such as processor and communication line faults. Our major concern here is software fault tolerance. We devote our attention to both dependability and efficiency, searching for a systematic way of using the minimum amount of hardware and time resources to attain the *required* software dependability. It is particularly emphasized that a quality fault-tolerant software should be obligated to behave as precisely required (e.g. by different delivery conditions), but it should not be liable for anything outside of the specified requirement.

Conditional utilization of redundancy requires extra time expenditure once a fault occurs. However, time is a limited resource in a real-time environment. The uncontrolled application of time redundancy can lead to a delay in the production of output information and will result in such information being classed as invalid if deadlines are missed. The maximum possible delay in our approach must be determined carefully according to the response time required. Related work exists in [Campbell et al 1979][Hecht 1976][Kim & Welch 1989]. Integration of fault tolerance and real-time issues is addressed thoroughly in [Bondavalli et al 1993]. It must be further mentioned that for applications where time redundancy is not acceptable, we have no choice but to sacrifice software efficiency to guarantee both timeliness and reliability, and full parallel (unconditional) execution of variants is therefore the only possible solution.

SCOP is designed intentionally as a general scheme for coping with both dependability and efficiency issues. In order to tolerate software faults (and some hardware-related faults), an instance of the SCOP scheme may employ an application-specific strategy for masking the effect of faults, such as multiple versions of software, diversity in data space, or simple retry of programs, depending upon special application requirements and considerations of cost effectiveness.

3.2.2 The SCOP Scheme

Basic Architecture

The SCOP scheme consists of a set of software variants, $V = \{v_1, v_2, ..., v_n\}$, an adjudication mechanism, and a controller that coordinates dynamic actions of the architecture. The main characteristics of SCOP include:

1) *Dynamic Use of Redundancy.* SCOP always tries to execute the least number of variants strictly necessary for providing a result which meets the stated delivery conditions. To do this it organises the execution of variants in phases, dynamically configuring a currently active set (CAS) V_i (a subset of V) at the

beginning of the *i*th phase. An adjudication is made after the execution of V_i in order to check if conditions for the release of a result are satisfied. The result will be output immediately and any further phases and actions will be ended once these conditions are met.

2) *Growing Syndrome Space.* A syndrome is a set of information used by an adjudicator to perform its judgement as to the correctness of a result. The syndrome information in SCOP is accumulated with the increase of phases. All the results produced and the additional information collected so far are employed to facilitate the selection of a correct result.

3) *Flexibility and Efficiency.* SCOP can be designed to obey different delivery conditions. Since the different conditions will usually have different fault coverages, SCOP is therefore able to provide different levels of dependability. Different conditions may be dynamically chosen by different applications, according to their degrees of criticality, or by the same application at different times, corresponding to different levels of graceful degradation. The initial CAS V_1 in the first phase is determined and could be changed with respect to different delivery conditions, while the set V_i in the *i*th phase ($i > 1$) can be constructed at run time based on information about the state of the system, so making efficient utilization of available resources.

4) *Generality.* Software variants used by SCOP can be designed using design diversity [Avezienis 1985], data diversity [Ammann & Knight 1988], or simple replication of a program [Huang & Kintala 1993] (when only transient software faults are expected).

The behaviour of SCOP can be described by the following control algorithm (shown in a Pascal-like notation with comments on the right side).

```
begin
i := 0;                                  {i is the index of the current phase, set to 0}
State_mark := G;                         {set current state as Go-state}
Si = {};                                 {set syndrome as empty}
C := one of { delivery conditions };     {set required delivery condition}
decide(max_phase);                       {based on timing constraints}
 while State_mark = G and                {while current state is Go-state and
 i < max_phase do                        current phase is less than maximum allowed}
   begin
   i := i+1;                             {start new phase}
   configure(C, Si-1, i, Vi);            {create new Currently Active Set}
   execute(Vi, Si);                      {execute and obtain new syndrome}
   adjudicate(C, Si, State_mark, res);   {set new state mark and select result}
   end;
if State_mark = E                        {current state is End-state or Failure state?}
   then deliver(res)
   else signal(failure);
end
```

The `decide` procedure determines the maximum number `max_phase` of possible phases to be permitted by the specified timing constraints. Procedure `configure` constructs the CAS set v_1 in the first phase according to the selected delivery condition and the given application environment, and establishes the CAS set v_i ($i > 1$) based on the syndrome s_{i-1} collected in the (i-1)th phase and the information on phases. The execution of a CAS may lead to a successful state `E`. Note that the variants in v_i are selected from those variants that have not been used in any of the previous phases, i.e. v_i is a subset of $v - (v_1 \cup v_2 \cup ... \cup v_{i-1})$. If the ith phase is the last one, v_i would contain all the remaining spare variants. The `execute` procedure manages the execution of the variants in CAS and generates the syndrome s_i, where s_0 is an empty set and s_{i-1} is a subset of s_i. Procedure `adjudicate` implements the adjudication function using the selected condition `c`. It receives the syndrome s_i, sets the new `State_mark` and selects the result `res`, if one exists. The `deliver` procedure delivers the selected result and the `signal` produces a failure notification.

> *Example*: Suppose that i) a distributed system composed of three processors is available for the parallel execution of up to three software variants; ii) seven software variants are provided; iii) the maximum time delay permitted is three phases, and iv) a result selected from at least three agreeing versions is considered as being deliverable. Here, $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$, $|V_1| = 3$, and $0 < i \leq 3$. Three examples of possible execution are illustrated in Table 3.2, where italics are used to indicate the agreeing results as seen by an ideal omniscient observer and bold characters are used to represent incorrect, disagreeing results.

Phase	V_i	Spares	Syndrome	Judgement & result
1	{v1,v2,v3}	{v4,v5,v6,v7}	*r1, r2, r3*	⇒ E, r1
1	{v1,v2,v3}	{v4,v5,v6,v7}	**r1, r2, r3**	G
2	{v4,v5}	{v6,v7}	**r1**, *r2*, **r3**, *r4, r5*	⇒ E, r2
1	{v1,v2,v3}	{v4,v5,v6,v7}	**r1, r2**, *r3*	G
2	{v4,v5}	{v6,v7}	**r1, r2**, *r3*, **r4**, *r5*	G
3	{v6,v7}	{}	**r1, r2**, *r3*, **r4**, *r5*, **r6, r7**	⇒ F

Table 3.2 Execution examples of SCOP

Figure 3.4 further gives a Petri-net execution model for the SCOP scheme, assuming that the maximum number of allowed phases is two. The example and the execution model demonstrate how SCOP reaches the required dependability in a dynamic manner when the availability of hardware resources is fixed. In practice, the amount of available resources and real-time constraints may vary. Particularly, in computing systems that function for a long period of time, the user may impose new requirements or modify the original requirements for timeliness, dependability etc. In addition, the

user may program more variants when the need arises. These uncertain factors require more complicated and more dynamic control and management. SCOP is further intended to cope with them.

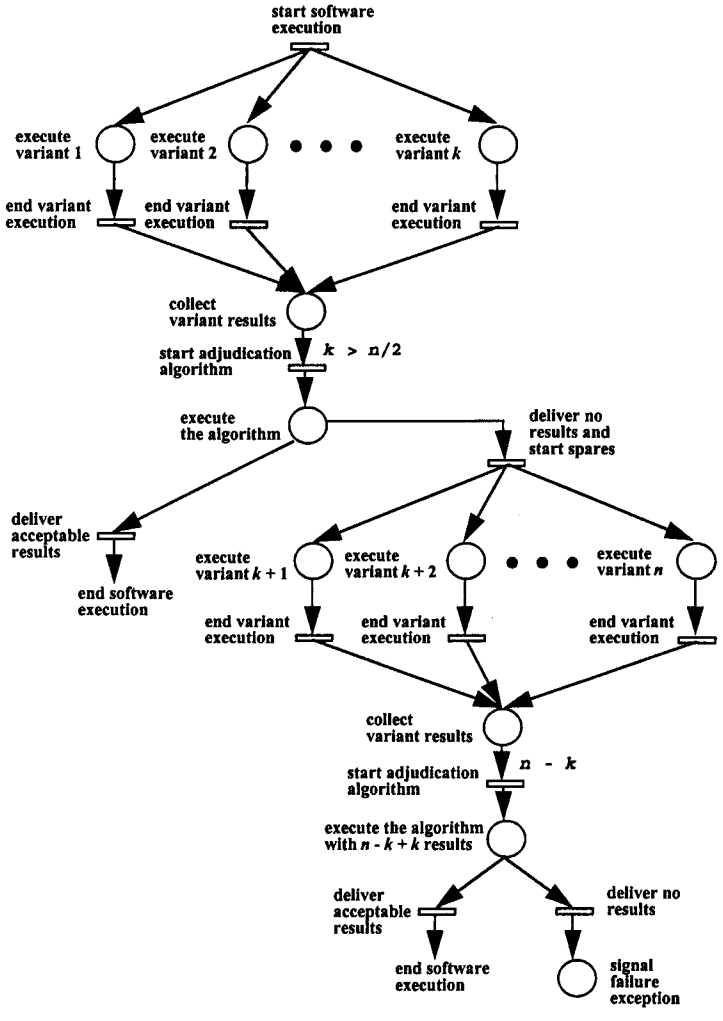


Figure 3.4 Execution model of the SCOP scheme

Dynamic Behaviour in a Distributed Computing Environment

In a large distributed system, hardware resources involving processors, memories and communication devices can be utilized by several competing concurrent applications, so that the amount of resources available for a specific application often varies. Furthermore, complex schemes for software fault tolerance may be necessary only for some critical part of the application which demands extra resources from the system. Dynamic management can make the allocation of resources more efficient. Let N_p be the maximum number of software variants which can be executed in parallel on hardware resources currently allocated to a given application, and T_d the time deadline that indicates the maximum response delay permitted for the application. Figure 3.5

illustrates a possible organization for SCOP and its dynamic behaviour in a varying environment.

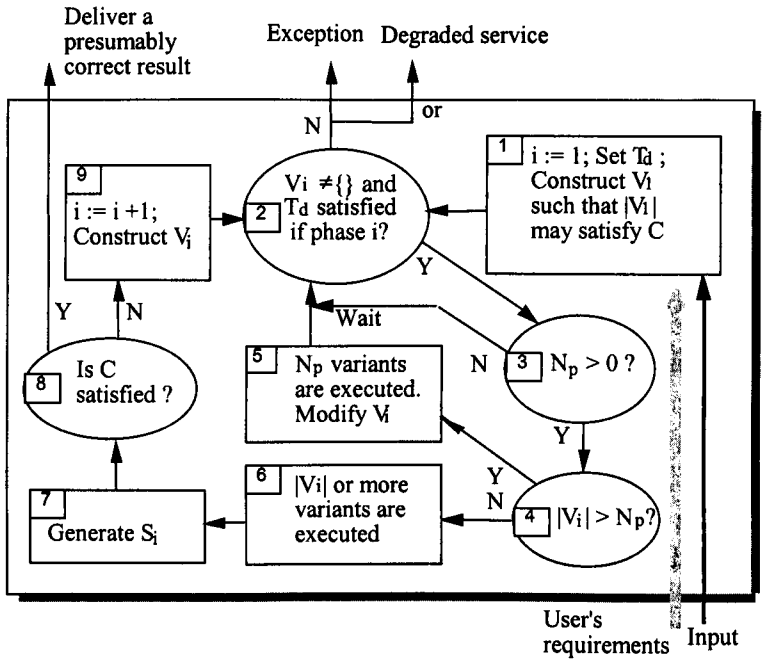


Figure 3.5 Dynamic behaviour of SCOP in a varying environment

Each step in the diagram is explained as follows.

1) Establish (for a given application) the delivery condition C (or several delivery conditions which permit different levels of dependability), according to the user's requirements, and then configure the initial CAS set, V_1, from V, which includes the minimum number of variants needed to be executed to generate a result satisfying the condition C in the absence of faults. Determine the timing constraint, T_d, based on the response requirement.

2) Check whether the time deadline T_d will be missed when the ith phase is initiated for the execution of V_i (i = 1, 2, 3, ...) and whether V_i is an empty set. If T_d allows for no new phase or V_i = { }, an exception will be raised to signal to the user that a timely result satisfying the required condition C cannot be provided. In this case, a degraded service may be attempted.

3) Check whether $N_p > 0$. $N_p = 0$ means that no variant can be executed at this moment due to the limitation of available resources in the system. Wait and go back to Step 2 to check T_d.

4) Check whether $|V_i| > N_p$. If $|V_i| > N_p$, only some of the variants in V_i can be executed within the current phase and thus additional time is needed for the execution of V_i.

5) Execute N_p software variants in V_i and modify V_i such that V_i excludes the variants which have been executed. Back to Step 2.

6) Since $|V_i| \leq N_p$, $|V_i|$ variants are executed and finished within the current phase. If the scheduler used by the supporting system allocates $N_p > |V_i|$ processors to SCOP during the ith phase, it is possible to consider the execution of N_p variants (more than $|V_i|$). This avoids wasting the resources that would be left idle otherwise, and requires the ability to select the additional variants among those not yet used.

7) Generate syndrome S_i based on all the information collected up to this point.

8) Check whether a result exists that satisfies the delivery condition C. If so, deliver the result to the user; otherwise Step 9.

9) Set $i = i + 1$ and construct a new CAS set, V_i, from the spare variants according to the information about the syndrome, the deadline and the resources available; if no sufficient spare variants are available, set V_i empty.

It is worth mentioning that the major purpose of this illustration is to outline the dynamic behaviour of SCOP. Practical applications will require specific designs. A design methodology for supporting this kind of dynamic management and control is developed in [Xu et al 1995b].

Implementation of SCOP

The main mechanisms needed to support a SCOP architecture are those which implement conditional execution of N software variants. Control of these variants is provided by a controller (similar to the driver program used in NVP). The controller is responsible for:

1) ensuring an identical set of input values to each software variant;

2) dynamically invoking an appropriate subset of variants;

3) waiting for the variants in the CAS set to complete their execution;

4) comparing the results produced by the variants and taking selective action.

We will examine below which existing mechanisms or measures used to implement NVP or RB can be employed directly for implementation of SCOP and which mechanisms need to be extended and improved.

Input Space. When executed, each variant must have access to an identical set of input values. (We ignore the possibility of data diversity for a moment.) There are two methods of implementing this: i) the controller communicates the set of input values to each variant, or ii) all variants access the input values from a shared, read-only, global

data structure. In practice, a large set of input values may result in heavy space and communication overheads. For *N*-version programming, the set of input values, or the data structure, can be updated using new values as soon as each variant receives or obtains the previous values. NVP also allows a variant to retain private data in structures local to its instantiation, to be used in subsequent executions of that variant, which could reduce the amount of input required. However, input values must be retained in the SCOP scheme until an execution of the scheme is completed, i.e. either the End-state *E* or the Failure-state *F* is reached. Since not all of variants are executed each time when the SCOP scheme is invoked, the variants must not retain data locally between executions, otherwise these variants could become inconsistent with each other. Generally speaking, SCOP may have a more significant input-space overhead than NVP though their requirements regarding the input mechanism are essentially identical. The cache mechanism used in recovery blocks [Anderson & Kerr 1976][Lee et al 1980] would provide a possibility of reducing the extra (both input and output) space cost imposed by SCOP.

Under a unified control framework provided by SCOP, data diversity (and message reordering) can be handily supported by a concrete implementation. Because the set of input values must be kept unchanged for each execution, once the execution of some spare variants is required, diverse input values to them can still be generated by diversifying the original, unchanged set of inputs [Ammann & Knight 1988] (or by reordering these input values [Huang & Kintala 1993].)

Synchronization. A mechanism is necessary for synchronizing the actions of the controller and the variants, and for communicating outputs from the variants to the controller. The variants in the CAS set wait and do not start processing until a start command is issued by the controller. Similarly, the controller waits until complete responses have been received from all the variants in the CAS set. The comparison check on the set of the results can then be performed. Some form of timeout mechanism is further required to deal with the situations that some variants do not complete their execution due to a design fault. Again, such a mechanism has no basic differences from that adopted by NVP except that the process of synchronizing the controller and the variants may be repeated (i.e. several execution phases) within an execution of SCOP. For example, the output vector method for communicating the results to the controller in NVP [Avizienis et al 1988b] can be applied directly to an implementation of SCOP.

Atomicity. Like NVP, the SCOP scheme also requires that each variant is executed atomically with respect to the other variants, namely, without any communication with or interference from the others. Multiprocessors and distributed computing systems would be ideal for achieving such atomicity since each variant can be executed concurrently on independent hardware.

Adjudication and Outputs. Similar to the requirements regarding input values, the information derived from the output values produced by previously executed variants

must be retained since any adjudication made by SCOP must be based on all the syndrome information collected so far. However, this information is not necessarily all the vectors of output values produced till now, but some representative vectors of results generated by the controller with the indication of the frequency of their appearance. The controller makes an adjudication on further actions, referring to data in the read-only data structure supporting run-time adjudication. Once the End-state E is reached, the selected vector of output values will be released, and the syndrome information and the set of input values for this execution will be discarded.

To summarize, the implementation of SCOP will not introduce serious technical difficulties, in comparison with the classical approaches such as NVP and RB. However, dynamic execution of software variants could cause extra space (e.g. memory) overheads for retaining related input and output data. When compared to the overall gains in hardware by the SCOP scheme (see the next evaluation section), these overheads should be minimal and not be of major concern in practice. (Table 3.3 compares SCOP with NVP and RB with respect to major implementation details.)

Scheme	Data Consistence	Execution Coordination	Execution Atomicity	Adjudication and Outputs
SCOP	Input mechanism similar to NVP, but retention of input values	Synchronization between the controller and variants for any execution phase	Isolation of separate variants supported by the hardware system	Retention of output values for growing syndrome space and dynamic adjudication
NVP	Control by the driver program or use of a global data structure	Synchronization between the driver and versions	Isolation of separate versions supported by the hardware system	Votes; outputs without retention (may retain local data for efficiency)
RB	Efficient recovery cache for retention of recovery data	Access to recovery cache and invocation to the spare alternate	Isolation of alternates by checkpointing and backward recovery	Acceptance tests; delivery of acceptable results without retention

Table 3.3 Major implementation characteristics of SCOP, NVP and RB

3.3 Analytic Evaluation of Fault-Tolerant Software

As discussed in Chapter Two, it cannot be guaranteed that independently designed software variants will fail independently, i.e. that faults in the different variants will occur at random and be unrelated. The dependability analysis of fault-tolerant software must therefore study the effect of related faults. A number of papers devoted to such dependability analysis have appeared in the literature (see related work addressed in Chapter Two). In particular, Arlat, Kanoun, and Laprie [Arlat et al 1990] developed complete fault classifications and presented a detailed evaluation of NVP and RB. Their analysis concentrated on basic architectures able to tolerate a single fault and

thereby the analytical conclusions can hold only for those specific instances. We augment published work by analyzing more general architectures that tolerate two or more software faults and by carefully identifying the ability of various approaches to tolerate independent and related faults. The results drawn from our analysis provide designers with richer information about the fault tolerance properties of various architectures than the results from traditional analysis, and show evidence that both $t/(n-1)$-VP and SCOP are a viable addition or alternative to present schemes for coping with software faults. Table 3.4 explains the notation to be used for the dependability evaluation.

Notation	
A_i	state of adjudicator's execution
B, C	state of [benign, catastrophic] failure caused by an undetected error
C_i	result comparator
C_X	Probability of catastrophic failure of approach X
D, U	state of [detected, undetected] failure
E	state of software execution
F_X	probability of failure of approach X
I	state of software idleness during the specified exposure period
N, n	number of software variants
p	probability that all variants produce the same correct results
q_I, q_A	probability of an independent fault in [a variant, the adjudicator]
q_{AD}, q_{AU}	probability of a [detected, undetected] independent fault in the adjudicator
q_{AV}	probability of related faults among the variants and the adjudicator
q_{mV}	probability of related faults among m variants
q_U	probability of an undetected failure
q_C	probability of a catastrophic failure due to an undetected error
r_i	result(s) of V_i
$R_X(t)$, $S_X(t)$	[reliability, safety] of approach X
V	State of variant execution
V_i	Ith software variant
σ	Departure rate from state I
$\omega_{i,i+1}$	(comparison) test outcome

Table 3.4 Notation for dependability evaluation

3.3.1 Evaluation of t/(n–1)-VP

We exploit the modelling framework in [Arlat et al 1990] for investigating the software redundancy needed to tolerate two or more faults and establish a slightly different model so that the different impacts of independent and related faults on software dependability can be examined. Three architectures are analysed that can tolerate at least two software faults: $t/(n-1)$-VP and NVP using five variants, the former adopting a simple diagnosis algorithm for result selection (see Table 3.1) and the latter employing the usual majority adjudication, and NSCP using six variants organized as three self-checking components. (Note that the NSCP architecture being considered here can tolerate two faults in most fault situations except related faults that occur in an active self-checking component.) Expressions for F_X and C_X, where $X \in \{t/(n\text{-}1)\text{-VP}, \text{NVP}, \text{NSCP}\}$, will be derived using a Markov approach.

Our analysis will be based on the following assumptions:

1) During the execution of the X scheme, related faults manifest themselves in the form of similar errors, whereas independent faults only cause distinct errors; and furthermore similar errors lead to common-mode failures, and distinct errors only cause independent failures;

2) all variants have the same probability of fault manifestation (or error);

3) only a single fault type, either independent or related, may appear during the execution of the scheme and no compensation [Laprie 1992] may occur between errors of the variants and of the adjudicator, i.e. either an error is detected or it causes an incorrect output;

4) probabilities of independent and related faults are significantly low such that the probability p can be approximated to 1 (as assumed by others in similar settings; see [Arlat et al 1990] for example).

These assumptions are used only to simplify the notation and the complexity in modelling and should not alter the significance of analytical conclusions. In particular, assumption 2) can be easily generalized to the case where the variants have respective fault characteristics. More complex models can be developed without applying assumption 4), i.e. probabilities of independent and related faults are allowed to be arbitrary (such models are described in [Tai et al 1993] and also used in the next section for the evaluation of SCOP.)

Dependability Model

We consider here two different but complementary attributes of dependability: continuity of service and non-occurrence of catastrophic failure. In general, we define software reliability as a measure of the time to failure and its safety as a measure of the

time to catastrophic failure [Arlat et al 1990][Laprie 1992]. The time (or the specified exposure period) in this definition is a relative concept and may mean a single run, a number of runs, or time expressed in calendar or execution time units of software. In the case of multiple runs, software may be idle between its executions. However software faults can manifest themselves only when software is executed. We will therefore focus on the execution process of software. Figure 3.6 shows a slight variation of the software behaviour model proposed by Arlat et al [Arlat et al 1990]. In this behaviour model, a detected failure (i.e. no service is delivered) is classified as *benign*; an undetected failure (i.e. an incorrect result is delivered) can be either benign, or *catastrophic*. Since several runs are possible, service delivery may be restored from benign failures. Note that transitions from D and B to I and from U to B or C are applied only to the safety evaluation. Based on a Markov approach to modelling, reliability of the X approach can be evaluated simply by:

$$R_X(t) = e^{-(\sigma Fx)t}$$

where t the specified exposure time (for a detailed discussion of this formula see [Arlat et al 1990]);

and safety by:

$$S_X(t) = e^{-(\sigma Cx)t}$$

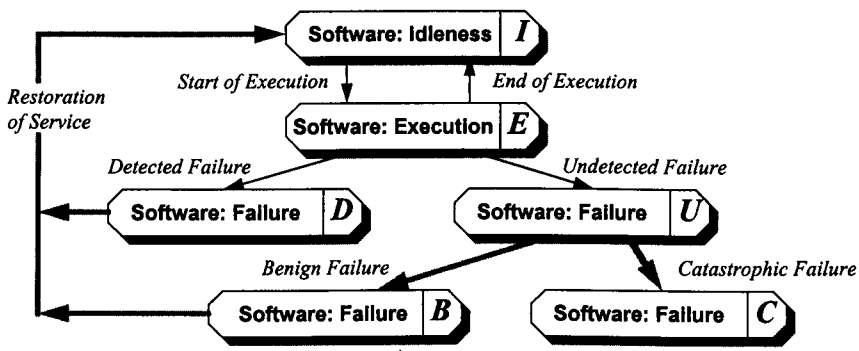


Figure 3.6 A modified behaviour model

A *t*/(*n*–1)-VP Model for 2/(5–1)-Architecture

Figure 3.7 describes a state-transition diagram for the 2/(5–1)-architecture based on the notation introduced in Table 3.4. From state E, execution states of the adjudicator are explained as follows.

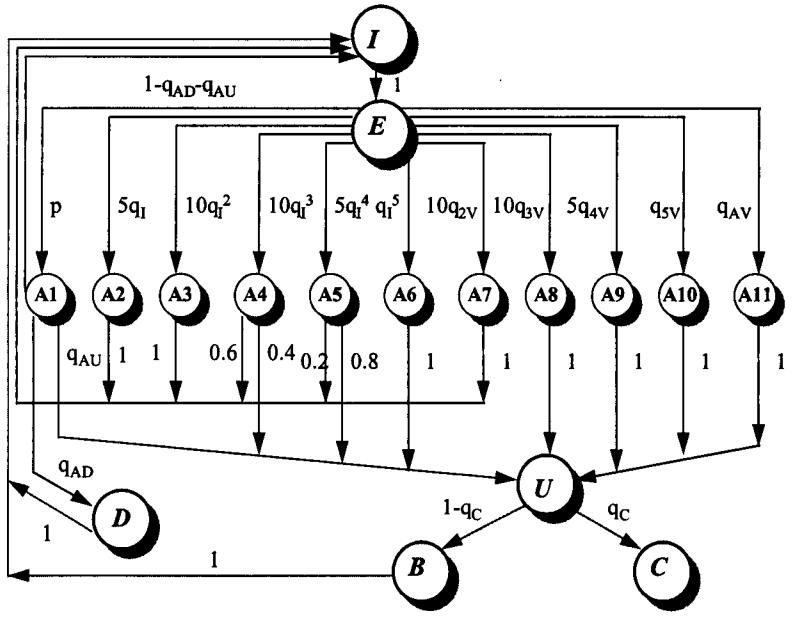


Figure 3.7 *t*/(*n*–1)-VP model

1) State $A1$ corresponds to the case in which five variants produce the same correct results. According to assumption 4), probability p that all variants produce correct results can be approximated to $1 - 5q_I - 10(q_I)^2 - 10(q_I)^3 - 5(q_I)^4 - (q_I)^5 - 10q_{2V} - 10q_{3V} - 5q_{4V} - q_{5V} - q_{AV}$ (≈ 1). Given no fault in any variant, different types of adjudicator failure will lead to states D and U with respective probabilities q_{AD} and q_{AU}.

2) States $A2$ and $A3$ indicate activation of one or two independent faults in variants given no related fault among the variants. These fault types can be tolerated by this 2/(5–1)-architecture.

3) States $A4$, $A5$ and $A6$ correspond to cases in which three or more independent faults manifest themselves in variants. Since the number of faults has exceeded the bound 2, these states may lead to a failure state. However, through a more precise analysis, it is found that *t*/(*n*–1)-VP can still deliver a correct result in some situations (see a further discussion below).

4) State $A7$ represents activation of related faults in any two variants. These faults can be tolerated.

5) States $A8$, $A9$ and $A10$ correspond to cases in which related faults manifest themselves in more than two variants, which are undetectable.

6) State $A11$ corresponds to activation of related faults between the adjudicator and the variants. This is also regarded as undetectable (see assumption 3).

In this 2/(5–1)-VP model, there is the transition from state $A4$ (or $A5$) to state I, that is, the architecture considered may still select a correct result as the system output even in the presence of more than two faults. Without loss of generality, take state $A4$ as an example. If three independent faults affect only three of variants V_1, V_2, V_3, and V_4, by assumption 1) their results will generate the syndrome where $\omega_{12} = \omega_{23} = \omega_{34} = 1$. The result of V_5 (a correct result) will then be chosen as the system output. Note that this class of events may occur with the probability $4q_I^3$. Similarly, if three independent faults affect only V_1, V_2 and V_5 (or only V_3, V_4 and V_5), according to Table 3.1, the selected result can be still a correct one, with the probability $2q_I^3$. To sum up, the conditional probability of the transition from state $A4$ to state I is $(4q_I^3 + 2q_I^3) / (10q_I^3) = 0.6$. Therefore, the transition from A4 to a failure state can actually take place with the conditional probability $(4q_I^3) / (10q_I^3) = 0.4$.

From the state-transition diagram of Figure 3.7, it follows that

$$F_{t/(n-1)\text{-}VP} = p(q_{AD} + q_{AU}) + 4(q_I)^3 + 4(q_I)^4 + (q_I)^5 + 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV}$$

A close but pessimistic approximation can be:

$$F_{t/(n-1)\text{-}VP} = q_{AD} + q_{AU} + 4(q_I)^3 + 4(q_I)^4 + (q_I)^5 + 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV} \quad (1)$$

For evaluation of safety, only state C is absorbing:

$$C_{t/(n-1)\text{-}VP} = q_C[q_{AU} + 4(q_I)^3 + 4(q_I)^4 + (q_I)^5 + 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV}] \quad (2)$$

An NVP Model for 5VP-Architecture

Figure 3.8 shows the NVP model for the 5VP-architecture.

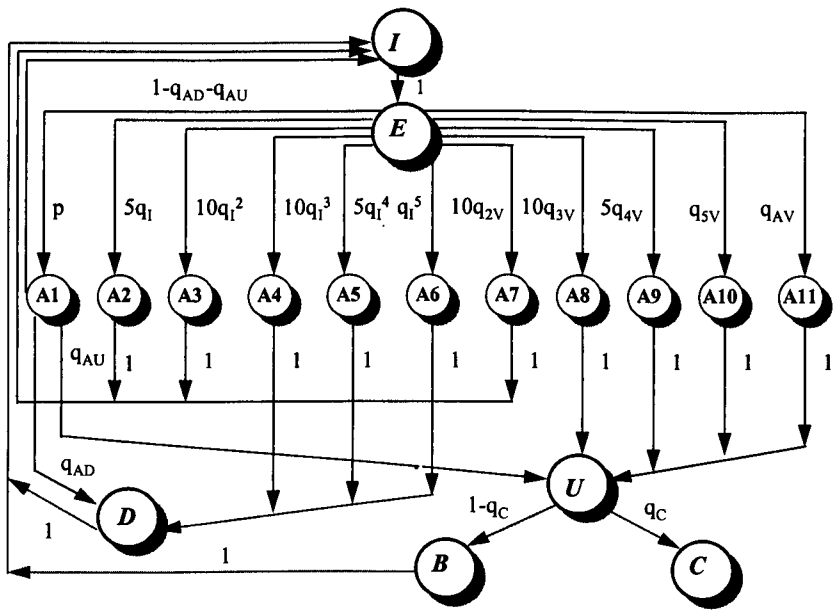


Figure 3.8 NVP model

The detailed analysis is essentially similar to that made for $t/(n-1)$-VP. A major difference is the case where multiple independent faults have an impact on three or more variants. In NVP, this case is much simpler — these faults will always lead to state D, assuming they are always detectable (but not tolerated). Thus, for reliability, F_{NVP} will be greater than $F_{t/(n-1)\text{-}VP}$:

$$F_{NVP} = q_{AD} + q_{AU} + 10(q_I)^3 + 5(q_I)^4 + (q_I)^5 + 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV} \quad (3)$$

However, due to the detectability of multiple independent faults we have for safety:

$$C_{NVP} = q_C[q_{AU} + 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV}] \quad (4)$$

which is obviously lower than $C_{t/(n-1)\text{-}VP}$.

An NSCP Model for 3SCP-Architecture

Figure 3.9 shows NSCP model for 3SCP-architecture. The interpretations of the states are similar to those of the $t/(n-1)$-VP model though there are thirteen states $A1 \sim A13$ to consider because of the use of six variants. Independent faults in one or two of variants can be tolerated. Independent faults in three or more variants can be either tolerated or detected, as indicated by states $A4$ and $A5$. This shows that the NSCP scheme is quite effective for the treatment of independent faults. However, cases where related faults manifest themselves among multiple variants become more complicated. On one hand, NSCP is not fault-tolerant in the worst case — any related faults in active self-checking components could lead to certain failure. On the other hand, some related faults can be tolerated or detected if they do not affect the pair of variants in an active self-checking component. Consider a representative case, state $A9$, in which related faults manifest themselves in three of the six software variants. There are three sub-cases to consider.

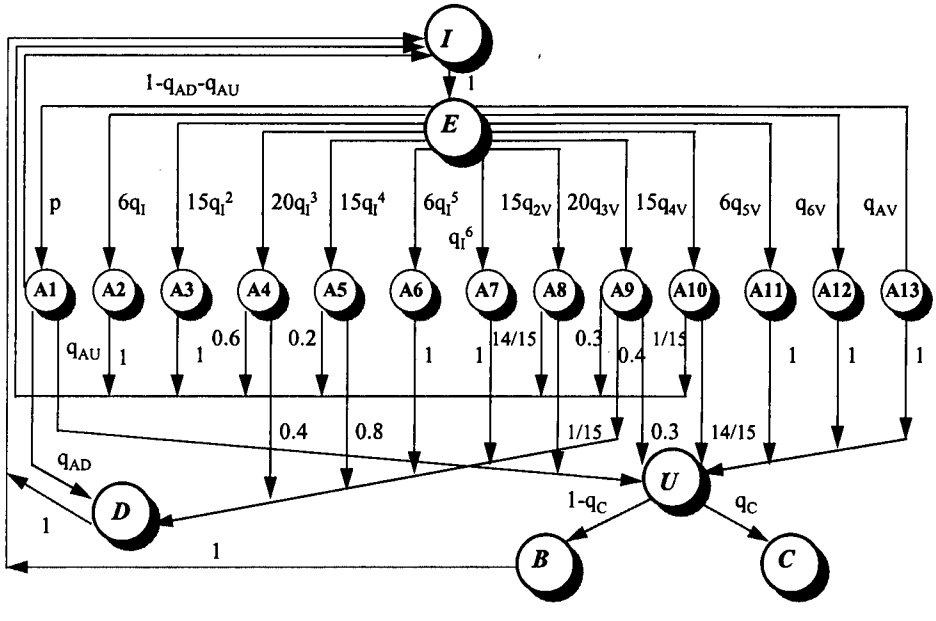


Figure 3.9 NSCP model

1) If related faults only occur in the spare self-checking components or such faults affect just a variant in the active component but not affect the first spare component, the 3SCP architecture can select a correct result and provide normal service. The conditional probability that this sub-case does occur is $[(6/20) \times (20q_{3V})] / (20q_{3V}) = 0.3$.

2) If related faults affect exactly one variant in every self-checking component, they can be detected effectively; the corresponding conditional probability is $[(2^3/20) \times (20q_{3V})] / (20q_{3V}) = 0.4$.

3) The worst sub-case is that related faults have an influence upon the pair of variants in the active component or an impact on the pair of variants in the first spare component given these related faults have affected a variant in the active one. In this sub-case, the 3SCP architecture will produce incorrect outputs, and the corresponding conditional probability is $[(6/20) \times (20q_{3V})] / (20q_{3V}) = 0.3$.

A similar analysis can be applied to other states. It therefore follows from the state-transition diagram of Figure 3.9:

$$F_{NSCP} = q_{AD} + q_{AU} + 8(q_I)^3 + 12(q_I)^4 + 6(q_I)^5 + (q_I)^6 + q_{2V} + 14q_{3V} + 14q_{4V} + 6q_{5V} + q_{6V} + q_{AV} \quad (5)$$

Since independent faults can be either tolerated or detected, safety of the NSCP architecture concerns only related faults:

$$C_{NSCP} = q_C[q_{AU} + q_{2V} + 6q_{3V} + 14q_{4V} + 6q_{5V} + q_{6V} + q_{AV}] \quad (6)$$

Remarks

Table 3.5 summarizes the specific expressions for q_I's and q_U's.

Parameters	$t/(n-1)$-VP	NVP	NSCP
$Q_{I:X}$	$4(q_I)^3 + 4(q_I)^4 + (q_I)^5$	$10(q_I)^3 + 5(q_I)^4 + (q_I)^5$	$8(q_I)^3 + 12(q_I)^4 + 6(q_I)^5 + (q_I)^6$
$q_{A:X}$	$q_{A(t/(n-1)\text{-VP})}$(comparators)	$q_{A(\text{NVP})}$(voter)	$q_{A(\text{NSCP})}$(comparator)
$q_{AV:X}$	$q_{AV(t/(n-1)\text{-VP})}$	$q_{AV(\text{NVP})}$	$q_{AV(\text{NSCP})}$
$q_{mV:X}$	$10q_{3V} + 5q_{4V} + q_{5V}$	$10q_{3V} + 5q_{4V} + q_{5V}$	$q_{2V} + 6q_{3V} + 14q_{4V} + 6q_{5V} + q_{6V}$
$Q_{U:X}$	$q_{AU} + 4(q_I)^3 + 4(q_I)^4 + (q_I)^5 + 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV}$	$q_{AU} + 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV}]$	$q_{AU} + q_{2V} + 6q_{3V} + 14q_{4V} + 6q_{5V} + q_{6V} + q_{AV}$

Table 3.5 Specific expressions for q_I's and q_U's

These expressions show that independent failures of the variants have a relatively small influence upon $t/(n-1)$-VP, but a larger impact on NVP and even more on NSCP. This is because the $t/(n-1)$-VP scheme possesses one of the significant characteristics of the

$t/(n-1)$-diagnosis technique; namely, it is possible in some fault situations for our proposed scheme to identify the correct results even though faulty variants are in the majority.

As assumed previously, the adjudicators used in the three specific architectures are: result comparison plus a diagnosis algorithm in $t/(n-1)$-VP, a voter in NVP, and result comparison (plus the result switch) in NSCP. According to their relative complexity, it would be reasonable to rank q_A's and q_{AV}'s as follows.

$$q_{A(\text{NSCP})} \leq q_{A(t/(n-1)\text{-VP})} \leq q_{A(\text{NVP})} \tag{7}$$

$$q_{AV(\text{NSCP})} \leq q_{AV(t/(n-1)\text{-VP})} \leq q_{AV(\text{NVP})} \tag{8}$$

It follows from Table 3.5 that related faults among variants have the same influence upon $t/(n-1)$-VP and NVP, but more serious on NSCP. This is a consequence of the fact that result comparison used in the self-checking components and the NSCP architecture itself are not effective enough to detect (or further tolerate) the related faults that may affect both variants in a self-checking component. Generally, this cannot be overcome by incorporating more variants into a given architecture. In contrast, both $t/(n-1)$-VP and NVP can tolerate some related faults under the same bound and furthermore their fault tolerance capability can be enhanced, at least in principle, by involving more software variants.

Summarizing, the analysis above could thus suggest the following general conclusions.

For reliability:

$$F_{t/(n-1)\text{-}VP} < F_{NVP} < F_{NSCP} \tag{9}$$

The inequality (9) means that the $t/(n-1)$-VP architecture has the lowest probability of failure — equivalently, the highest reliability. Due to high detectability of independent faults, NVP is however less sensitive to undetected faults than $t/(n-1)$-VP. The probability $q_{U(t/(n-1)\text{-VP})}$ for $t/(n-1)$-VP looks relatively high since this scheme may fail to detect some independent faults when the bound on the number of faulty variants is violated. This probability could be reduced by using more software variants. Note that the probability $q_{U(\text{NSCP})}$ is high as well, but again the incorporation of more variants would have no effect on safety enhancement of NSCP. So for safety:

$$C_{NVP} < C_{t/(n-1)\text{-}VP} \leq C_{NSCP} \tag{10}$$

Finally, it is important to notice that the evaluation data obtained here was used only to uncover the relative advantages and disadvantages of these schemes under consideration. For a given design using a particular scheme, the evaluation results also show how the design could be modified to further improve its dependability. Since the notion of software dependability captures many different concerns, including reliability, availability, safety and security, our analysis demonstrates the need of a delicate balance between these complementary attributes. In practice, a software

designer must make a decision as to which technique is likely to be most appropriate for a specific application.

3.3.2 Evaluation of SCOP

We analyze the SCOP scheme based on an architecture that makes it comparable with other schemes. The SCOP architecture chosen involves three variants and the associated delivery condition requires that at least two variants produce the same results. It executes just two variants in the first phase. The assumptions 1), 2) and 3) discussed in Section 3.3.1 are still applicable here. However, in the interest of general applicability, probabilities of independent and related faults are allowed to be significantly high in the following analysis, with the upper bound up to one.

Due to such adjustments to basic assumptions, the complexity in modelling and state space will increase. We therefore consider three architectures that are simpler than those used in our $t/(n-1)$-VP analysis. The NVP architecture uses three variants based on the usual majority adjudication, the RB architecture consists of one primary block and one alternate block, and NSCP contains four variants organized as two self-checking components. Expressions for F_X and C_X, where $X \in$ {NVP, RB, NSCP}, will be derived using a Markov approach.

We condition the probability q_A on a conservative base, namely, the fault will always cause the adjudicator to reject a result (or a majority) given the result (or the majority) is correct, or the adjudicator to output a result given the result is incorrect and no majority exists. Note that a comparison-based adjudicator is normally application-independent. We will no longer consider a related fault between the variants and the adjudicator in the models for SCOP, NVP and NSCP, but only in the model for RB which manifests with probability q_{AV}. For reasons of simplicity, it is assumed that an independent fault in the SCOP adjudicator will only manifest itself at the end of the final adjudication.

Finally, we will use a simplified version of the model explained in Figure 3.6, as illustrated in Figure 3.10.

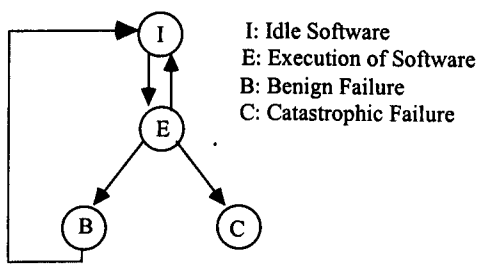
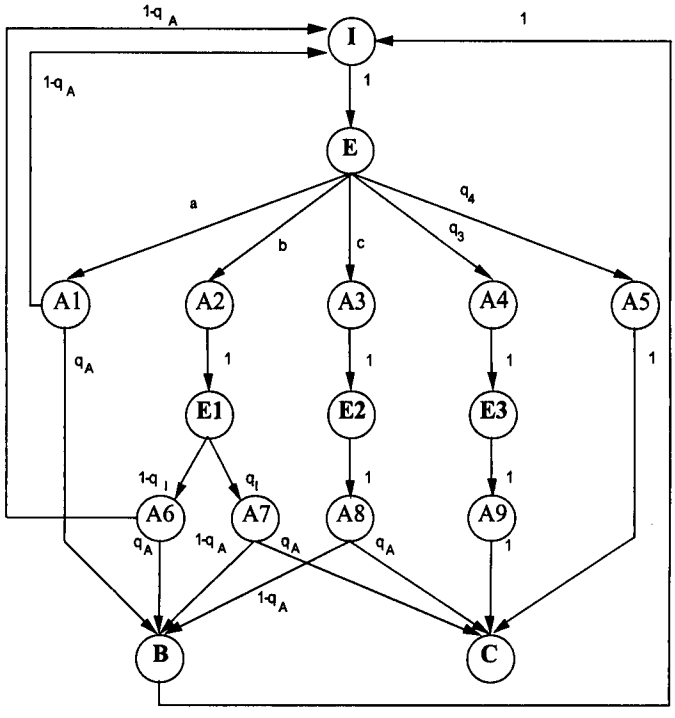


Figure 3.10 A simple behaviour model

A SCOP Model for 3VP-Architecture

A detailed model for SCOP is now constructed as follows. In Figure 3.11 state E is defined as the execution state of variants in Phase One and states $E1$, $E2$ and $E3$ as the execution states in Phase Two. States from $A1$ to $A5$ and from $A6$ to $A9$ are defined as the execution states of the adjudicator in Phases One and Two respectively.



$$a = (1 - q_1 - q_2)(1 - q_3 - q_4),\ b = q_1(1 - q_3 - q_4),\ c = q_2(1 - q_3 - q_4)$$
$$q_1 = 2q_I(1 - q_I),\ q_2 = q_I^2,\ q_3 = 2q_{2V},\ q_4 = q_{2V} + q_{3V}$$

Figure 3.11 SCOP model

Execution states of the Adjudicator in Phase One:

1) State $A1$ corresponds to the case in which two active variants produce the same correct results.

2) State $A2$ corresponds to the case in which an independent fault in one of the active variants causes an incorrect result.

3) State $A3$ corresponds to the case in which two independent faults in the active variants cause two disagreeing results.

4) State $A4$ corresponds to the case in which a related fault between one of the active variants and the third variant causes two disagreeing results.

5) State $A5$ corresponds to the case in which a related fault causes the same, but incorrect, results.

Execution States of the Variants in Phase Two:

1) State $E1$ corresponds to the case in which an independent fault in one of the active variants causes the second phase.

2) State $E2$ corresponds to the case in which two independent faults in the active variants cause the second phase.

3) State $E3$ corresponds to the case in which a related fault between one of the active variants and the third variant causes the second phase.

Execution States of the Adjudicator in Phase Two:

1) State $A6$ corresponds to the case in which the third variant produces a correct result given a fault in one of the active variants.

2) State $A7$ corresponds to the case in which an independent fault in the third variant occurs given a fault in one of the active variants.

3) State $A8$ corresponds to the case in which the third variant produces a correct or an incorrect result given two independent faults in the active variants.

4) State $A9$ corresponds to the case in which a related fault occurs between one of the active variants and the third variant.

From the state-transition diagram of Figure 6, we obtain that:

$$F_{SCOP} = (q_2 - 2q_A q_2 + q_I q_1 - 2q_A q_I q_1 + q_A)(1 - q_3 - q_4) + C_{SCOP} \quad (11)$$

$$C_{SCOP} = q_A(q_I q_1 + q_2)(1 - q_3 - q_4) + q_3 + q_4 \quad (12)$$

where $q_1 = 2q_I(1 - q_I)$, $q_2 = q_I^2$, $q_3 = 2q_{2V}$, and $q_4 = q_{2V} + q_{3V}$.

For the purpose of comparison with the NVP architecture, let $q_i = 3q_I^2(1-q_I)+q_I^3$ and $q_r = 3q_{2V}+q_{3V}$. It follows that

$$F_{SCOP} = q_A(1 - q_i)(1 - q_r)+(1 - q_A)q_i(1 - q_r) + C_{SCOP} \quad (13)$$

$$C_{SCOP} = q_A q_i(1 - q_r) + q_r \quad (14)$$

Following a similar approach to dependability modelling, we conclude that:

$$F_{NVP} = q_A(1 - q_i)(1 - q_r) + (1 - q_A)q_i(1 - q_r) + C_{NVP} \quad (15)$$

$$C_{NVP} = q_A q_i(1 - q_r) + q_r \quad (16)$$

$$F_{RB} = q_A(1 - q_I - q_{2V} - q_{AV}) + q_A q_I(1 - q_I) + (1 - q_A)q_I^2 + q_{2V} + C_{RB} \quad (17)$$

$$C_{RB} = q_A q_I^2 + q_{AV} \quad (18)$$

$$F_{NSCP} = q_A(1 - q_{iv})(1 - q_{rv}) + (1 - q_A)q_{iv}^2(1 - q_{rv}) + 4q_{2V} + C_{NSCP} \quad (19)$$

$$C_{NSCP} = q_A q_{iv}^2(1 - q_r) + q_{rv} - 5q_{2V} \quad (20)$$

where $q_{iv} = 2q_I(1 - q_I) + q_I^2$ and $q_{rv} = 6q_{2V} + 4q_{3V} + q_{4V}$.

From these specific expressions for probabilities of benign and catastrophic failures, we could claim that SCOP can provide the same level of dependability as NVP. However the probability q_A is generally different with respect to different architectures. The SCOP adjudicator we have examined will be invoked twice if some faults are detected in the first phase. It is therefore possible that SCOP delivers an incorrect result in the early phase or rejects a correct majority, starting a new phase, due to a manifestation of a fault in the adjudicator. When such fault situations are taken into account, the dependability of SCOP would be slightly lower than that of NVP with a simple majority voter. In fact, probabilities associated with the adjudicator of NVP can also vary significantly since various complicated adjudicators may be employed [Di Giandomenico & Strigini 1990].

RB seems to be the best, but an acceptance test or AT is usually application-dependent. The degree of design diversity between an AT and the variants could be different with respect to different applications so that q_{AV} (i.e. the probability of a related fault between an AT and the variants) may vary dramatically. Moreover, the fault coverage of an AT is an indicator of its complexity, where an increase in fault coverage generally requires a more complicated implementation.

Consumption of Resources

Let T_X be the time necessary for the execution of a complete phase in the X scheme, and f be the maximum number of variant failures to be tolerated by the scheme. Note that T_X consists of the execution time of the variants and the time for both adjudication and control. We now conduct an analysis of resource consumption, referring to more general architectures: 1) the SCOP architecture involves $2f + 1$ variants, with the delivery condition that requires at least $f + 1$ identical results, executing $f + 1$ variants in the first phase, 2) the NVP architecture uses $2f + 1$ variants based on the usual majority adjudication, 3) the RB architecture consists of one primary block and f alternate blocks; and 4) NSCP contains $2(f + 1)$ variants organized as $(f + 1)$ self-checking components.

In the interests of concentrating on efficiency, we assume no timing constraints on the service and perfect adjudicators. Table 3.6 reports some results about resource consumption, in which each cell of the NoVariant column indicates the total number of variants needed to be executed when a scheme attempts to complete its service, in direct proportion to the amount of hardware resources. Both the worst and the average case are shown. In the worst case, SCOP requires the amount of hardware resources

that supports the execution of $(2f + 1)$ variants, but on an average it needs hardware support just for the execution of the $(f + 1)$ variants.

Scheme	NoVariant worst	NoVariant average	TIME worst	TIME average
SCOP	(f+1)+f	≅(f+1)	$T_{SCOP}+fT_{SCOP}$	$\cong T_{SCOP}$
NVP	2f+1	2f+1	T_{NVP}	T_{NVP}
RB	1+f	≅1	$T_{RB}+fT_{RB}$	$\cong T_{RB}$
NSCP	2(f+1)	2(f+1)	T_N+ft_{switch}	$\cong T_N$

t_{switch} is the time for switching the self-checking components

Table 3.6 Comparison of resource consumption

SCOP will terminate any further execution if $(f + 1)$ agreeing results are obtained in the first phase, i.e. the condition for delivering a result is satisfied. This scenario happens when i) a related fault manifests itself and affects all the $(f + 1)$ variants; or ii) all the variants produce the same correct results. Events like the failure or success of individual variants are usually not independent but positively correlated under the condition that all the variants are executed together on the same input [Knight & Leveson 1986]. This factor determines that the probability of observing the "event ii)" would be higher than what might be expected assuming independence. Let p_V be the probability that a single variant gives a correct result. We know:

Probability that SCOP would then stop at the end of the first phase >

Probability that the f+1 variants would produce correct results > p_V^{f+1}

Experimental values of p_V such as reported in [Knight & Leveson 1986a] are sufficiently high so that we would claim SCOP almost always gives the same fast response as NVP or even faster, as will be discussed later. Note that the worst case, in which SCOP operates with the longest execution time $T_{SCOP} + fT_{SCOP}$, has in fact a very rare probability of occurrence. It occurs only when the phase ends with f agreeing results and every remaining variant, assigned to run in one of phases from Two to $(f + 1)$, produces a different result. It is therefore reasonable to rank SCOP as more efficient than NVP and NSCP. RB seems to be better again, relying to some extent on the use of acceptance tests. However, acceptance tests for RB are often difficult to devise and, in many cases, they will provide no guarantee that a variant has executed correctly.

If timing constraints for delivering the result are considered, for example, if the maximum number of allowable phases is p where $p \leq f + 1$, the SCOP's worst case of execution time will become $p \times T_{SCOP}$. Note, however, that this limitation on the number of phases heavily impacts the average usage of variants only if $p = 1$ (in this case the execution of all the variants is required). Otherwise the first phase, very likely the only one, always involves just $(f + 1)$ variants. The basic RB scheme will not be

applicable directly to the case that $p < f + 1$, as RB needs $f + 1$ phases to deal with successive manifestations of f faults. Parallel implementations of RB may be suitable [Kim 1984], but they operate at the cost of more variants executed within a phase.

More precisely, the execution times of various adjudication functions (and control algorithms) can be significantly different with respect to specific fault coverages and algorithm complexity. Furthermore, the execution times of variants in a scheme can differ because of the requirements for design diversity and equivalent variant functionality. Since in SCOP the subsequent phases will be utilized much less than the first phase, the faster variants (those which correspond to more effective implementations and whose execution times are shorter) may be chosen in the first phase. The penalty caused by variant synchronization (that requires the system to wait for the slowest variant) can be thus reduced, as compared with NVP.

Example: (*Comparison of SCOP and NVP*) Table 3.7 gives the related figures of resource consumption in SCOP and NVP where $T_{SCOP} \leq T_{NVP}$ (i.e. $T_{SCOP1} \leq T_{3VP}$, $T_{SCOP2} \leq T_{5VP}$, and $T_{SCOP3} \leq T_{7VP}$) and $p_V = (1 - 10^{-4})$ (which is the average reliability of the variants derived from the experiment in [Knight & Leveson 1986a]). Data for SCOP have been obtained based on the assumption that just two phases are allowed. The table shows that SCOP consumes almost the same amount of time as NVP to provide services, but it requires just the amount of hardware resources as that which supports $(f + 1)$VP (that only resists at most $f/2$ variant failures), rather than $(2f + 1)$VP.

	Scheme	No. of variants executed (Average)	Time consumption (Average)
Average Cost for General Case	SCOP	$(f+1)+(1-p_V^{f+1})f$	$[1+(1-p_V^{f+1})]T_{SCOP}$
	NVP	$2f+1$	T_{NVP}
N = 3 f = 1	SCOP	2.0002	$1.0002T_{SCOP1}$
	NVP	3	T_{3VP}
N = 5 f = 2	SCOP	3.0006	$1.0003T_{SCOP2}$
	NVP	5	T_{5VP}
N = 7 f = 3	SCOP	4.0012	$1.0004T_{SCOP3}$
	NVP	7	T_{7VP}

Table 3.7 Resource consumption of SCOP and NVP

3.4 Empirical Comparison

We have conducted an experiment in order to examine the effectiveness of both $t/(n-1)$-VP and SCOP in comparison with traditional schemes such as NVP and RB. This experiment was not supposed to study the effectiveness of fault-tolerant software in general, but was particularly designed to investigate the relative advantages and disadvantages of the two advanced techniques developed in this chapter. We used GNU

C++ version 2.6.3 and a target environment composed of a set of workstations running UNIX and connected through TCP/IP (see Figure 3.12).

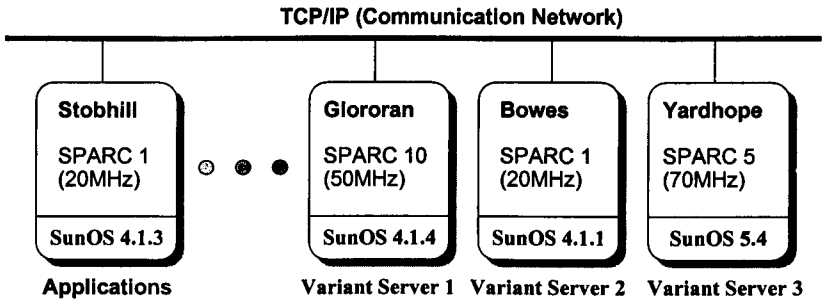


Figure 3.12 Target environment (hardware/software)

The target distributed system follows a client-server manner. Clients and servers communicate each other by means of TCP/IP BSD sockets. We define a special `Socket` object that provides the functionality of communication through sockets. The `Socket` object has two operations: `ConnectWrite()` that is responsible for establishing the connection between the client and the server and for passing an object onto the server, and `ReadClose()` responsible for receiving the result (or object) from the server and for closing the connection. Because communication costs are not our major concern, we assume that the supporting communication mechanisms are highly dependable and effective enough for our experimentation. Other communication mechanisms are actually possible for our research, such as remote procedure calls (RPC) supported by the Arjuna system [Shrivastava et al 1991].

3.4.1 C++ Implementation of t/(n–1)-VP and SCOP

A large sorting application was implemented using four variations of the basic sorting algorithm. This implementation consists of an application program (as the client) and three sorting servers. The client and servers are located on different processing nodes. First, the client program passes an object that contains a random list of integers onto every sorting server. The servers then sort the list and send the results back to the client.

The object passed between the client and servers is of type `arrayList`, associated with a set of operations that can be re-defined and implemented in an alternative manner by inheritance. Take the operation `Sort` as an example. The normal `Sort` operation may be based on a simple "bubble sort" algorithm. Sub-classes of class `arrayList` can be then implemented as faster versions of sorting, e.g. `quickArrayList`, `heapArrayList`, and `shellArrayList`, or as more dependable versions, e.g. `t/(n-1)-VPArrayList`, `SCOPArrayList`, `RBArrayList`, and `NVPArrayList` based on *t*/(*n*–1)-VP, SCOP, RB and NVP respectively. Figure 3.13 shows the corresponding class hierarchy.

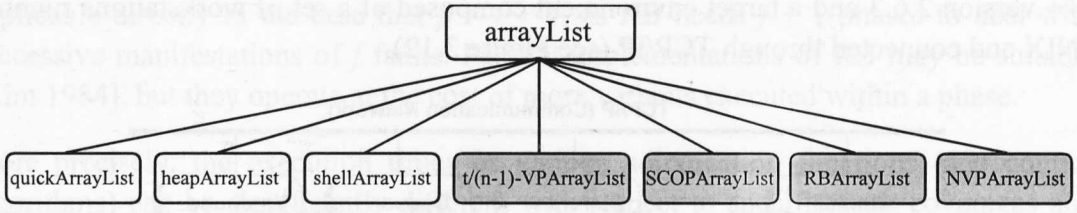


Figure 3.13 Class hierarchy of array sorting

Control mechanisms for different software fault tolerance schemes are implemented as a collection of C++ library programs. A `SFTClass` contains various operations that implement *t*/(*n*–1)-VP, SCOP, RB and NVP.

```
template <class T> class SFTClass {
    public:
        SFTClass();
        void Variant(int port, char *machine);
            bool t/(n-1)-VP(T& obj);
            bool SCOP(T& obj);
            bool RB(T& obj);
            bool NVP(T& obj);
        private:
            // internal states of SFTClass
};
```

Program 3.1 Class `SFTClass`

In Program 3.1, `T` is the class/type of the object that is passed to remote sorting servers and manipulated by the sorting variants. The `port` and `machine` parameters of the `Variant()` operation indicate the addresses of remote servers that actually run sorting variants through the control of a software fault tolerance scheme. The control operation such as `t/(n-1)-VP(T& obj)` and `SCOP(T& obj)` returns a boolean result indicating whether an acceptable result has been obtained (`TRUE`) or not (`FALSE`). Program 3.2 shows an implementation of the *t*/(*n*–1)-VP scheme.

```
template <class T>
bool SFTClass<T>::t/(n-1)-VP(T& obj)
{
    Socket<T> s[MAXVARIANT];
    T auxL[MAXVARIANT];
    for (int i=0; i<nvar; i++)          // activate all variants
        s[i].ConnectWrite(port[i], machine[i], obj);
    for (int i=0; i<nvar; i++)
        s[i].ReadClose(auxL[i]);
    if (obj.t/(n-1)-diagnostor(auxL))
        return(TRUE);                   // acceptable result found
    return(FALSE);                      // no acceptable result found
}
```

Program 3.2 Implementation of *t*/(*n*–1)-VP

More precisely, the `t/(n-1)-VP` operation receives an object from the client and passes it to all the remote variants. When the results are returned from the variants, a $t/(n-1)$-diagnostor (i.e. a $t/(n-1)$-fault diagnosis algorithm) is executed. If an acceptable result is identified, boolean value `TRUE` is returned, or otherwise `FALSE` is returned indicating that the $t/(n-1)$-VP scheme fails to find an acceptable result. In a slightly different way, a two-phase SCOP architecture is implemented as follows.

```
template <class T>
bool SFTClass<T>::SCOP(T& obj)
{
        Socket<T> s[MAXVARIANT];
        T auxL[MAXVARIANT];
        // start phase one
        for (int i=0; i<nvar/2+1; i++)
                s[i].ConnectWrite(port[i], machine[i], obj);
        for (int i=0; i<nvar/2+1; i++)
                s[i].ReadClose(auxL[i]);
        if (obj.Voter(auxL))
                return(TRUE);
        // start phase two
        for (int i=nvar/2+1; i<nvar; i++)
                s[i].ConnectWrite(port[i], machine[i], obj);
        for (int i=nvar/2+1; i<nvar; i++)
                s[i].ReadClose(auxL[i]);
        if (obj.Voter(auxL))
                return(TRUE);
        return(FALSE);
}
```

Program 3.3 Implementation of SCOP

The `SCOP` operation first accepts an object from the client and then starts phase one by passing the object to a subset of sorting variants executed in parallel. When results are returned from the subset, they are applied to a voter. If an acceptable result is found, `TRUE` will be returned, or otherwise phase two has to be started. A new subset of variants are executed concurrently and their results are voted again together with the results obtained in the previous phase. `TRUE` is returned if an acceptable result is found in the end or `FALSE` is returned indicating that no acceptable result is achieved. More details as to the use of these reusable classes will be discussed in Chapter Five when we address the issues associated with supporting systems.

3.4.2 Timing Results Based on Software-Fault Injection

In order to examine the behaviour and relative effectiveness of various implementations testing based on software-fault injection was performed. The execution time is measured in microseconds (μseconds), assuming that the time of network communication is constant. A data set of 200 elements was used to measure times with

or without injected faults in programs. Table 3.8 reports the testing results from our experiment, where "✓" means that a correct result is obtained by a variant or a fault tolerance scheme for a given testing round.

		Experiment Set One	Experiment Set Two	Experiment Set Three	Experiment Set Four
Quick Sort Variant One	Result	✓	fail	fail	fail
	Time	3196	397	397	3319
Heap Sort Variant Two	Result	✓	✓	fail	✓
	Time	12037	12037	9688	12037
Shell Sort Variant Three	Result	✓	✓	✓	✓
	Time	2788	2788	2788	2788
$t/(n-1)$-VP Implementation	Result	✓	✓	✓	✓
	Time	13746	13746	11391	13746
SCOP Implementation	Result	✓	✓	fail	✓
	Time	13741	21660	19300	21660
RB Implementation	Result	✓	✓	✓	fail
	Time	5403	15464	19574	5669
NVP Implementation	Result	✓	✓	fail	✓
	Time	17163	17163	14379	17163

Table 3.8 Effectiveness and performance-related testing results

We start from the normal situation without fault injection (i.e. experiment set one) in order to examine typical run-time overheads of $t/(n-1)$-VP, SCOP, RB and NVP. In this case, every `Sort` operation was executed normally, and all lists returned are correctly sorted. Fault-tolerant versions based on various schemes generally have longer execution times than faster versions of the normal sorting operation. This is mainly due to the required coordination between variants and the execution of the adjudicator. RB uses `QuickSort` as its first alternate and executes a simple acceptance test that examines the sorted list. Its execution time is therefore slightly longer than that of the `QuickSort` operation. However, because other three schemes have to wait for the completion of the slowest `HeapSort` variant before being able to execute their adjudication functions, they need longer execution times than the `HeapSort` operation. It is interesting to notice that both $t/(n-1)$-VP and SCOP (that executes only Phase One in this case) have faster adjudicators than NVP that uses a voter to compare the results from different variants.

The situation that a single fault occurs in one variant is considered in experiment set two. Various software faults are injected into the `QuickSort` server, e.g. by randomly commenting a line of the code. All faults are detected successfully by $t/(n-1)$-VP, SCOP and NVP. There is little change in the execution times of $t/(n-1)$-VP and NVP,

but a significant increase in SCOP. This is simply because SCOP has to perform Phase Two in this case and then votes all the results from different variants. An increase is also observed in the execution time of RB since the second variant (i.e. `HeapSort`) must be invoked. Some faults injected can pass the acceptance test used in RB. Such a situation will be addressed in experiment set four later.

Experiment set three specifies the situation that independent faults occur in two different variants. Software faults are injected into both `QuickSort` and `HeapSort`. When $t/(n-1)$-VP and RB are still able to deliver a correct solution, both SCOP and NVP fail to identify an acceptable result since there exists no a majority of the sorted lists. In this case, $t/(n-1)$-VP selects the result of `ShellSort` as correct via $t/(n-1)$-diagnosis. However, it may make a wrong decision. It is observed that $t/(n-1)$-VP delivers an incorrect result when faults are injected into `ShellSort` instead of `HeapSort`. For certain applications, an acceptance test may be needed on the result of `ShellSort` for the consideration of safety. For this multiple fault situation, RB has to activate all variants in turn to reach a correct decision, with a longest execution time of 19574 microseconds. In the worst case that a related fault injected into both `QuickSort` and `HeapSort` causes identical incorrect sorting, $t/(n-1)$-VP, SCOP and NVP all pick up a wrong solution. It is only RB that survives this type of fault.

However, there are some types of faults that cannot be detected by the acceptance test used in RB. When RB delivers an incorrect result in this case, all the other three schemes mask this type of faults very effectively, with run-time overheads similar to those observed in case two. Figure 3.14 illustrates the impact of the number of faulty variants upon the execution time of various fault tolerance schemes. The overhead of fault tolerance is also indicated with respect to the fastest `ShellSort` operation. Both $t/(n-1)$-VP and NVP have much less varying execution times. In particular, $t/(n-1)$-VP imposes the least run-time overhead on an average.

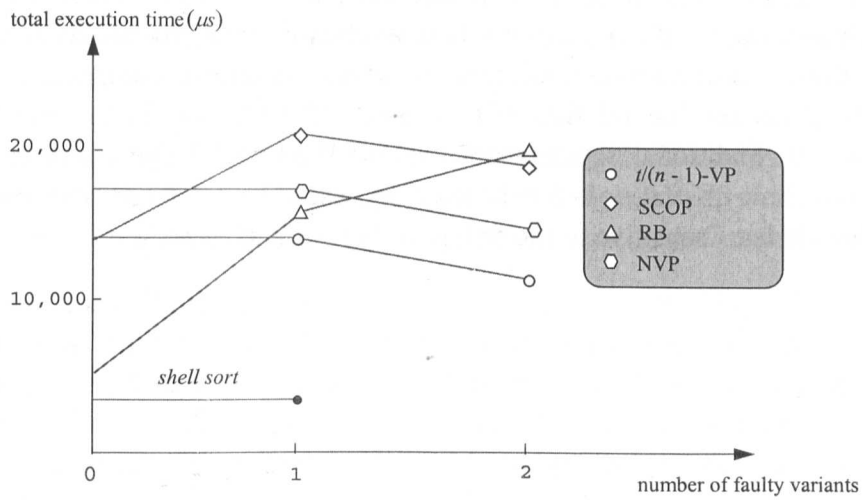


Figure 3.14 Impact upon execution times of fault-tolerant software

3.5 Summary

Building fault-tolerant software has been studied intensively and many schemes have been devised, especially for sequential programs. After years of investigation and development, there are still unsolved problems related to this topic and new schemes are still being developed and refined for domain-specific applications. In this chapter we have developed two advanced techniques for improving certain aspects of fault-tolerant software. The $t/(n-1)$-VP scheme is aimed at increasing the reliability of some common techniques such as N-version programming with majority voting. A well-developed fault diagnosis technique used in hardware was employed to simplify the adjudication mechanism of $t/(n-1)$-VP and to precisely identify correct results based on a limited set of testing information. When $t/(n-1)$-VP has the common ability of tolerating multiple related faults between variants, it is in general more reliable than many existing techniques. In particular, under certain circumstances, $t/(n-1)$-VP can deliver correct service even if the majority of variants have produced incorrect results. It also permits possible forms of graceful degradation.

SCOP is a hybrid technique that combines some advantages of both NVP and RB. It improves efficiency aspects of NVP-type schemes by providing adaptation of resource consumption to varying failure characteristics, while at the same time achieving the degree of dependability similar to that permitted by NVP and its variations. In addition, from the implementation viewpoint SCOP will not introduce serious technical difficulties although extra memory space is required for retaining related input and output data.

Our analytic evaluation based on a Markov approach generally supports the above conclusions. Unlike existing work, this dependability analysis has considered more complicated architectures than usual and examined intentionally the ability of various schemes to *tolerate* related faults between variants or related faults between the adjudicator and variants. It demonstrates how a scheme could provide certain degree of protection from common-mode failures. A small empirical comparison was also presented to illustrate the relative effectiveness of both $t/(n-1)$-VP and SCOP in comparison with traditional schemes such as NVP and RB. The collected data and timing results show that $t/(n-1)$-VP scheme imposes the least run-time overhead in general, which is least sensitive to the failure of the variants as well.

Chapter 4

Fault Tolerance in Concurrent Object-Oriented Software

Concurrent and distributed computing systems often give rise to complex asynchronous and interacting activities. The provision of fault tolerance becomes very difficult in such circumstances [Randell 1984]. One way to control the entire complexity, and hence facilitate error recovery, is to somehow restrict interaction and communication between concurrent activities. Atomic actions are the usual tool employed in both research and practice to achieve this goal (see the discussion in Chapter Two).

The use of object-oriented (OO) techniques is also considered as a very promising approach for building complex fault-tolerant software. This is because such techniques are based on many well-established software engineering principles such as data abstraction, encapsulation, modularity, hierarchies, and strong typing, thereby assisting complexity control and promoting clear system structuring. In fact, many new architectural developments in the area of large distributed systems are, to some extent, object-based or object-oriented.

As a generalized form of the basic atomic action structure, the concept of *Coordinated Atomic Actions* (or CA actions) was developed in 1995 [Xu et al 1995a], especially for concurrent/distributed object systems. CA actions provide a general mechanism for enclosing complex interaction activities and facilitating error recovery. However, this concept itself was still evolving: in contrast to its part related to the control of cooperative activities, of which a good understanding had already been obtained, several important problems including exception handling and software fault tolerance had to date not been resolved satisfactorily or had not been studied extensively.

In this chapter we will first revisit the CA action concept, following the description of an abstract model for concurrent/distributed systems. Significant aspects and properties of CA actions are described formally based on a linear-time temporal logic system [Lamport 1994][Manna & Pnueli 1991]. By using CA actions as a basic unit of error confinement, we then address issues of handling exceptions in complex object systems. Particular attention is paid to exceptions that occur in the environment of a computing system, and to exceptions that occur simultaneously in different nodes of a distributed system. Finally, we will investigate how errors could be recovered within the CA

action framework based on the use of the design diversity approach to provide software fault tolerance.

4.1 Abstract Model for Concurrent/Distributed Systems

This section first introduces a simple abstract model that characterizes concurrent/distributed systems, and then uses a suitably realistic example to explain key elements of the model.

4.1.1 Objects, Threads and Actions

We are concerned with software systems, which may be implemented on a variety of hardware. A system is viewed here as a set of interacting objects. An object is a named entity that combines a data structure (internal state) with some associated operations; these operations determine the externally visible behaviour of the object. In general, computation execution results in invocations on some object operations, possibly updating their internal state. It is usually assumed that, in the absence of concurrent invocations and failures, the invocation of an operation will produce consistent state changes to the object.

A thread is an agent of computation, and an active entity that is responsible for executing a sequence of operations on objects. A thread can exist syntactically as a powerful control abstraction or as a purely run-time concept. A system is said to be concurrent if it contains multiple threads that behave as though they are all in progress at one time. In a distributed or parallel computing environment, this may be literally true — several threads may execute at once, each on its own processing node.

An action is also a control abstraction that allows the application programmer to group a sequence of operations on objects into a logical execution unit. Actions may be associated with some desirable properties. During their execution, a variety of commit protocols are required to enforce corresponding properties. For example, atomic (trans)actions have the properties of atomicity, consistency, isolation, and durability (see the related discussion in Chapter Two) and can be used to ensure consistency of shared objects even in the presence of failures and concurrent access. CA actions further emphasize the enclosure of multi-thread cooperation and the strict prohibition of information smuggling into or out of the action boundaries [Xu et al 1995a]. This property facilitates complex error recovery that may involve multiple concurrent threads.

In order to perform effective concurrency control, it is crucial to identify different forms of concurrency properly. In [Hoare 1978], a set of concurrent processes is classified as being in one of three categories, namely independent, competing or cooperating. This classification is much related to the passing of information between processes. We will discuss and use a slightly different classification and focus more on the sharing of common objects.

Concurrent threads are said to be independent if the sets of objects accessed by each thread are disjoint — a trivial case. Competitive concurrency arises when concurrent threads that are designed independently for different computations have access to a set of common objects but have to compete for these objects. Cooperative concurrency arises when several threads are designed collectively and invoked concurrently to perform a pre-defined computation. Access to a set of common objects from these cooperative threads is specially ordered according to application-specific requirements. In reality, different kinds of concurrency may co-exist in a complex application and thus require a general mechanism for concurrency control.

Practical systems may be structured using the notion of objects and actions, or objects and threads, or a combination of all three abstractions. To further explain this, we will examine a realistic system below.

4.1.2 A Realistic Example: The Arjuna System

Arjuna is an object-oriented programming system, originally implemented in C++ and recently re-implemented in Java, that provides a set of tools for the construction of fault-tolerant distributed applications. The Arjuna research effort began at Newcastle in late 1985. Since then, the system has been fully tested and implemented to run on networked UNIX systems, and has been used for building a number of applications including the Newcastle University Student Registration system [Parrington et al 1995], distributed database systems, fault-tolerant parallel computing over a network of workstations, and Internet applications [Little & Shrivastava 1998]. Arjuna system software has been freely available for research, development and teaching purposes since 1992, and its Java version, named JavaArjuna, has been productized in order to be commercially available (`http://arjuna.ncl.ac.uk`).

The Arjuna system uses objects as the main repositories for holding the system state, and supports nested atomic actions for constructing distributed applications. Distributed computing is achieved by invoking operations on objects which may be remote from the invoker. Such a remote operation is performed via a remote procedure call (RPC). By ensuring that objects are persistent and only manipulated within an atomic action, it can be guaranteed that the integrity of objects, and hence the integrity of the system, is maintained even in the presence of failures (e.g. hardware node crashes in a distributed system) and concurrent access to objects.

Figure 4.1 illustrates the hardware architecture used in the Newcastle University Student Registration system and the organization of the Arjuna software. In particular, when not in use a persistent object is held in an object store (i.e. a stable object repository) and is activated on demand, when an invocation is made, by loading its state and operations from the persistent object store to the volatile store, and associating with it an object server for receiving RPC invocations. In addition, the name server keeps identification and location information about persistent objects,

while atomic transaction modules provide an application-level interface for controlling operations on objects.

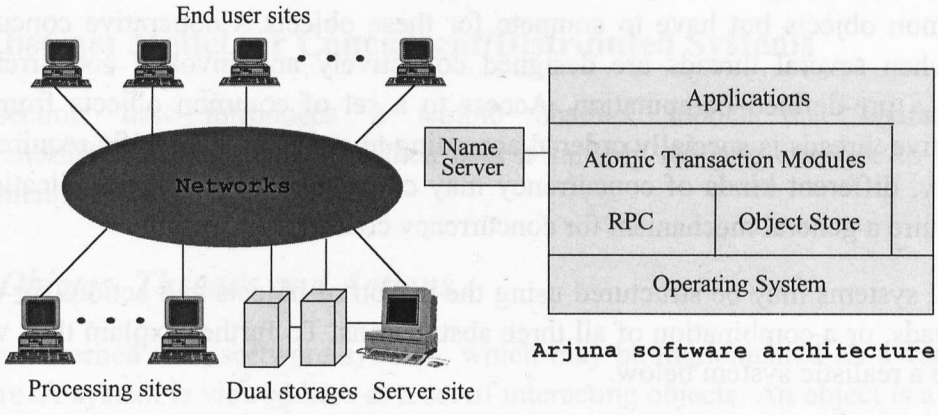


Figure 4.1 The Arjuna system

The original Arjuna system corresponds precisely to the object and atomic action model [Shrivastava et al 1993]. However, JavaArjuna adds the multi-threaded mechanism to the original system and provides application programmer with an additional abstraction to control complex concurrency. The thread that first arrives at an atomic action interface starts a transaction, and other subsequent threads will be able to join the same transaction later. The participating threads may leave the transaction asynchronously during the execution of the action, but the last one must be responsible for ending the action. In the JavaArjuna system, competitive concurrency is well controlled by the atomic transaction structure, and cooperative concurrency may be achieved by appropriate use of the Java multi-threaded control mechanism.

4.2 Coordinated Atomic Actions Revisited

In this section, the CA action concept is first described informally using a simple example. Its major properties are then formalized based on temporal logic. The main purpose of developing a formalization of CA actions here is to help clarify the CA action concept itself and to promote better understanding of several complicated and confusable aspects. In particular, a careful and formal treatment is given to enclosure-related properties that characterize how CA actions achieve strict error confinement. The formal description of the CA action concept indeed facilitates and promotes the correct use of CA actions as a structuring tool in our subsequent design and development of an actual CA action-based system (see Chapter Five).

4.2.1 Informal Description and an Example

Given the model for concurrent/distributed systems that was introduced in Section 4.1, a CA action is an abstract mechanism for coordinating multi-threaded interactions and ensuring consistent access to shared objects. It also presents a general scheme for achieving fault tolerance by combining conversations, transactions and exception

handling into a uniform structuring framework. Three main characteristics of the CA action scheme are:

1) *Multi-Threaded Coordination and Enclosure.* A CA action provides a logical enclosure of a group of operations on a collection of objects. Within the CA action, these operations are actually performed cooperatively by one or more *roles* executing in parallel. The interface to a CA action specifies both the objects that are to be manipulated by the action and the roles that are to manipulate these objects. In order to perform a CA action, a group of concurrent threads must come together and agree to perform each role of the action, with each thread undertaking its appropriate role. They enter and leave the action synchronously. Two forms of concurrency, cooperative and competitive, are permitted by CA actions. Roles of a CA action that have been designed collectively cooperate with each other in order to achieve certain joint and global goals, but they must interact only within the boundaries of the CA action. If the objects that are to be manipulated by a CA action are external to the action, they may be shared (or competed for) with other actions concurrently. Any access to such objects from concurrent actions must satisfy certain atomicity conditions based on appropriate concurrency control protocols so that the external objects cannot be used as an implicit means of "smuggling" information [Kim 1982] into or out of an action.

2) *Fault Tolerance.* If an error is detected inside a CA action, appropriate forward and/or backward recovery measures will be invoked cooperatively in order to reach some mutually consistent conclusion. To perform forward error recovery, a CA action must provide an effective means of coordinating the use of exception handlers. To perform backward error recovery, a recovery line must be associated with a CA action, which coordinates the recovery points of the objects and threads participating in the action so as to avoid the *domino effect* [Randell 1975]. An acceptance test can and ideally should be provided in order to determine whether the outcome of the CA action is successful. Based on appropriate recovery protocols, a CA action, when accessing its external objects, must satisfy the failure atomicity condition [Lynch et al 1993] so that individual error recovery or abortion of the CA action can be performed alone without affecting other concurrent actions and threads accessing its external objects.

3) *Multiple Outcomes.* The desired effect of performing a CA action is specified by a set of post conditions and can be checked by an acceptance test with respect to these conditions. The effect only becomes visible if the test is passed. The acceptance test allows both a normal outcome and one or more exceptional (or degraded) outcomes, with each exceptional outcome signalling a specified exception to the surrounding environment. The CA action is considered to have failed if the action failed to pass the test, or roles of the action failed to agree about the outcome. In this case, it is necessary to undo the potentially visible effects of the CA action and signal an `abort` exception to the surrounding environment. If the CA action is unable to satisfy the "all-or-nothing " property

(e.g. because the undo fails), then a `failure` exception must be signalled to the surrounding environment indicating that the CA action has failed to pass its acceptance test and that its effects have not been undone. (The system has probably been left in an erroneous state and this must be dealt with by the enclosing CA action, assuming there is one, e.g. some external perhaps manual strategy.) As shown in Figure 4.2, a given performance of a CA action will only produce one of the following four forms of outputs: a normal outcome, an exceptional outcome, an `abort` exception, or a `failure` exception.

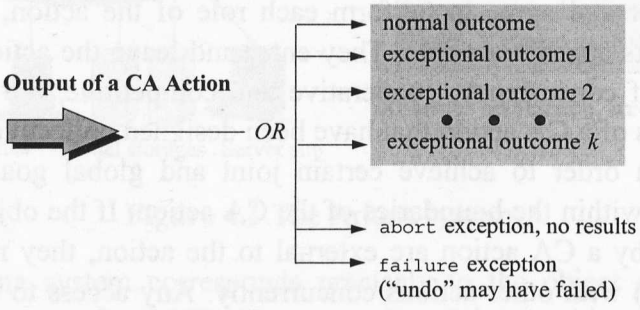


Figure 4.2 Possible outcomes of a CA action

Example 4.1: Figure 4.3 shows an example in which two concurrent threads enter a CA action in order to play the corresponding roles. Within the CA action two concurrent roles communicate with each other and manipulate the external objects cooperatively in pursuit of some common goal. However, during the execution of the CA action, an exception `e` is raised by one of the roles. The other role is then informed of the exception and both roles transfer control to their respective exception handlers H_1 and H_2 for this particular exception, which attempt to perform forward error recovery. The effects of erroneous operations on external objects are repaired by putting the objects into new correct states so that the CA action is able to exit with an acceptable outcome. Two threads leave the CA action synchronously at the end of the action. (As an alternative to performing forward error recovery, the CA action could undo the effects of operations on the external objects, roll back and then try again, possible using diversely designed software alternates.)

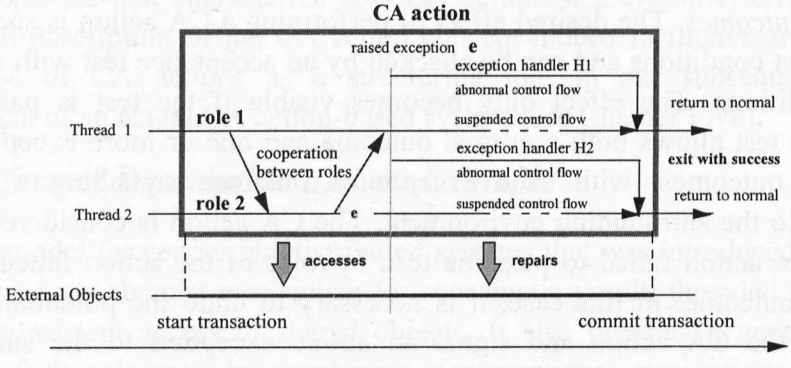


Figure 4.3 Example of a CA action

4.2.2 Temporal Logic

Following the informal description and the example of CA actions presented in the last section, we will discuss the CA action concept in more detail and in a precise manner. To do so we require some high-level formalization of CA actions. Such formalization should eventually permit us to formally verify properties of systems designed using CA actions. This is the major reason that we decide to use a logic-based approach, e.g. linear temporal logic, rather than a more process-oriented formalism.

A linear-time temporal logic system is chosen here as a specification language for specifying and proving properties of the CA action concept. Lamport [Lamport 1994], and Manna and Pnueli [Manna & Pnueli 1991] give a detailed discussion of how to specify and refine programs and their properties within a temporal logic framework. We summarize those aspects needed for formalizing properties of CA actions.

The syntax of temporal logic formulae extends that of formulae in ordinary first-order predicate logic by introducing several temporal operators such as $\Diamond$ (eventually), $\Box$ (henceforth), and $\underline{\Diamond}$ (sometime in the past). In general, a logic formula is constructed from variables, functions, predicates, the separator $\bullet$, the boolean operators $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), and $\Rightarrow$ (implication), and the temporal operators. The temporal operators have higher binding power than the boolean operators and quantifiers; parentheses may be used whenever the need arises. The variables appearing in a temporal logic formula may refer to program variables or logic variables.

Temporal logic formulae are interpreted over, in general infinite, sequences of program states. Such sequences are generated by the execution of a program, e.g. possible execution sequences of a CA action instance a. Each state assigns values to the program variables, and so an execution sequence can be viewed as a trace of the values of the program variables. A formula with the form $\Box P$ is true in a state of an execution sequence if P is true in that state and in all subsequent states of the sequence. A formula with the form $\Diamond P$ is true in a state of an execution sequence if a successor state of that state in the sequence exists in which P is true. By contrast, $\underline{\Diamond} P$ is true if a predecessor state exists in which P is true. In fact, most formulae that specify properties of CA actions are of the form $\Box P$, i.e. they specify properties that are always required to hold henceforth (in the literature frequently referred to as safety properties or invariants [Schneider 1997]).

The concurrent execution of several programs can be implemented by interleaving the execution sequences of these programs. However, such interleaving is in general nondeterministic, but different interleaved sequences may exhibit the same overall behaviour. A temporal formula describing a property of a concurrent system can therefore characterize all execution sequences for which the formula holds. Given a CA action instance, a temporal formula P is said to be a property of the action instance if all possible execution sequences of it satisfy P.

Let C be the set of action instances in a given context. When it causes no confusion, we will use the phrase "action c" instead of "action instance c". The set *roles*(c) contains all roles of action c. There are also two important sets of objects: *ex-objects*(c) is the set of external objects of action c which can be shared with other actions and threads, and *objects*(s) is the set of objects on which the operation sequence s is performed, where s may be interpreted as a role or as a single operation.

For any $c \in C$, we consider two basic predicates: *begin*(c) is true in a state where action c begins, while *end*(c) is true in a state where action c ends. These two predicates characterize the *initial* state and the *final* state in an execution sequence of a CA action.

4.2.3 Elementary Properties of CA Actions

We will now formally describe some elementary properties of CA actions, including properties that characterize relative orders of action-related execution states, that specify action entrances, exits and possible outcomes with respect to pre- and post-conditions, and that are related to the nesting of actions. More advanced properties such as enclosure and fault tolerance will be addressed in subsequent sections.

First, it follows from the definition of *begin*(c) and *end*(c) that a CA action c that ends must have begun earlier.

$$\forall c \in C \bullet \Box\ (end(c) \Rightarrow \underline{\Diamond}\ begin(c)) \qquad \text{(P1)}$$

A CA action c that begins will end at some time in the future.

$$\forall c \in C \bullet \Box\ (begin(c) \Rightarrow \Diamond\ end(c)) \qquad \text{(P2)}$$

We need to define two new predicates in order to specify certain interface properties: *called*(r) is true in the state of an execution sequence where role r is called by a thread (typically undertaking a role of an enclosing action), and *return*(r) is true in the state where r returns to its caller.

If a CA action c begins, all of its roles must have been called earlier.

$$\forall c \in C \bullet \Box\ (begin(c) \Rightarrow \forall r \in roles(c) \bullet \underline{\Diamond}\ called(r)) \qquad \text{(P3)}$$

If a CA action c ends, all of its roles must return in due course.

$$\forall c \in C \bullet \Box\ (end(c) \Rightarrow \forall r \in roles(c) \bullet \Diamond\ return(r)) \qquad \text{(P4)}$$

Consider four categories of possible outcomes of an action c: normal outcomes, exceptional outcomes, no outcome (when the action is aborted) and outcomes caused by failure. With respect to different outcome categories the final state *end*(c) can be further classified into four mutually exclusive sub-states: *normalend*(c), *exceptionalend*(c), *abortedend*(c), and *failedend*(c). In order to characterize these sub-

states, we distinguish between pre-conditions, different types of post-conditions, and testing conditions: *pre*(c) is true in the state where the pre-condition holds on the values of external objects of c, *n-post*(c) is the post condition associated with the normal outcome, and *e-post*(c) is true in the state where at least one of the post-conditions corresponding to different exceptional outcomes is satisfied. In addition, *accept*(c) is true in the state where the values of external objects of c pass the acceptance test.

If a CA action c began with the satisfied pre-condition and ends normally, its normal post-condition should hold.

$$\forall\, c \in C \bullet \Box\, (\underline{\Diamond}\, (begin(c) \wedge pre(c)) \wedge normalend(c) \Rightarrow n\text{-}post(c)) \qquad \text{(P5)}$$

If a CA action c began with the satisfied pre-condition and ends exceptionally (with a signalled exception), then one of its exceptional post-conditions should hold.

$$\forall\, c \in C \bullet \Box\, ((\underline{\Diamond}\, (begin(c) \wedge pre(c)) \wedge exceptionalend(c)) \Rightarrow e\text{-}post(c)) \qquad \text{(P6)}$$

If a CA action c ends with abortion, all its effects on the external objects should have been undone. (Let V_o be the set of possible values of an object o and *value*(o) be a function that returns o's value in a specified state.)

$$\forall\, c \in C,\, o \in ex\text{-}objects(c),\, o_i \in V_o \bullet$$

$$\Box\, (\underline{\Diamond}\, (begin(c) \wedge value(o) = o_i) \wedge abortedend(c) \Rightarrow value(o) = o_i) \qquad \text{(P7)}$$

If a CA action c ends with failure, the values of external objects can be arbitrary. No meaningful post-condition can be given in this worst case.

Acceptance tests are usually regarded as an implementation technique, and can be used in practice to examine whether the post-conditions will really hold. Ideally, we can design an acceptance test that provides a necessary and sufficient check, but in practice one often has to settle for a test that is necessary but not sufficient, i.e.

$$\forall\, c \in C \bullet \Box\, (n\text{-}post(c) \vee e\text{-}post(c) \Rightarrow accept(c)) \qquad \text{(P8)}$$

Nesting of CA actions allows a CA action instance to be composed of several, possibly concurrent, actions and operations, but initial and final states of all related actions must be properly ordered. Now let *parent*(d) be a function whose return value is the parent, or enclosing action, of a nested CA action d. It is important to notice that a nested action d begins only after its enclosing action c has begun; and c ends only after d has ended earlier.

$$\forall\, c, d \in C \mid c = parent(d) \bullet \Box\, (begin(d) \Rightarrow \underline{\Diamond}\, begin(c)) \qquad \text{(P9)}$$

$$\forall\, c, d \in C \mid c = parent(d) \bullet \Box\, (end(c) \wedge \underline{\Diamond}\, begin(d) \Rightarrow \underline{\Diamond}\, end(d)) \qquad \text{(P10)}$$

4.2.4 Enclosure, Unidirectional Enclosure and Non-Enclosure

The enclosure property is one of the most important properties of a CA action, and it emphasizes failure atomicity [Lynch et al 1993] of the CA action and is also relevant to execution atomicity. More precisely, this property means:

1) Any form of information can be passed into or out of a CA action only at the entrance and exit of the action, and

2) during the execution of an action, i.e. between the initial state and the final state of the action, a role inside the action cannot interact in any way with a role or thread that is not in the action.

In order to state this property formally, we first introduce an additional predicate: *inf_pass*(o, c) is true in the state where some information is passed from or to action c via object o. For a given CA action, its external objects are the only means of passing information from or to the action. This is because the effect of a CA action can be observed only through the state (or values) of its external objects. If *inf_pass*(o, c) is true in some state during the execution of c, excluding the initial state and final state of c, then i) other actions or outside threads observe the state change of c's external objects made by c after it has begun, or ii) c observes the state change of its external objects made by other actions or outside threads after c has begun. The enclosure property of a CA action c implies that neither of the above two cases can occur, i.e.

$$\forall\, c \in C, o \in \textit{ex-objects}(c) \bullet \Box\ (\textit{inf\_pass}(o, c) \Rightarrow \textit{begin}(c) \vee \textit{end}(c)) \qquad (P_e)$$

For a given c, it follows immediately that P_e holds if a particular implementation guarantees that c is executed as a whole with respect to its external objects, and no other actions or outside threads can access these objects during the execution of c. P_e also guarantees that any error recovery or abortion of a CA action can be performed alone without affecting any other actions and threads at the same level of nesting.

However, this is too restrictive and allows for little concurrent sharing of external objects. In practice, by carefully interleaving the operations of other actions and/or threads on the external objects of c, it is possible to increase concurrency and improve performance, but P_e must be relaxed. We will now address two forms of relaxation of P_e in order to achieve a greater degree of (competitive) concurrency.

Unidirectional Enclosure

Define the predicate *change*(o, c) to be true in the state of an execution sequence of action c where c changes the state of object o. Consider a constraint on concurrent sharing of objects: if a CA action c changes the state of an external object o, then that object o can be accessed by other actions or threads only after c has ended or aborted. This constraint allows a CA action c to observe the state change of an external object o made by other actions as long as c does not change o during its execution (i.e. only

partial or unidirectional enclosure is imposed, and certain information may be passed into a CA action during its execution.)

$$\forall c \in C, o \in ex\text{-}objects(c) \bullet \Box\ (inf\_pass(o, c) \Rightarrow$$

$$(begin(c) \vee end(c) \vee (\neg\ (\underline{\Diamond}\ change(o, c)\ \vee \Diamond\ change(o, c)))))$$

(C0)

However, condition C0 still ensures that error recovery or abortion of a CA action can be performed alone without affecting other actions and threads. This is because under this condition a CA action only has to recover those objects that it has changed. Note that such change has not yet been seen by the outside. Therefore, error recovery or abortion of the action will have no impact on other actions and threads.

Non-Enclosure

It is obvious that if more information is allowed to pass into or out of an action, i.e. to allow a greater degree of interleaving of operations, further concurrency will be obtained. However, this will greatly complicate the task of performing error recovery or abortion since cascading recovery or aborts of concurrent actions may be required. Moreover, the consistency property must be always maintained even though the isolation property is relaxed. We will derive two conditions that ensure the consistency of shared objects despite interleaved access to the objects.

Let C_p be the set of action instances at the same level of nesting within a given enclosing action p, i.e. $C_p = \{a \in C \mid p = parent(a)\}$. We first define three relationships between actions and roles at a given level of nesting.

For any actions $c, d \in C_p$, the execution of action c is said to *indivisibly precede* the execution of action d if the following holds:

$$\forall o \in ex\text{-}objects(c) \cap ex\text{-}objects(d), \circ_i \in V_o \bullet$$

$$\Box\ ((begin(d) \wedge value(o) = \circ_i) \Rightarrow \underline{\Diamond}\ (end(c) \wedge value(o) = \circ_i))$$

We use $c \gg d$ to represent the execution sequences where c indivisibly precedes d.

Two actions $c, d \in C_p$ are said to be *concurrent* if there is always a state in their execution sequences where both actions are performing their own operations, that is

$$\Box\ (\neg\ (\Diamond\ begin(c) \vee \underline{\Diamond}\ end(c) \vee \Diamond\ begin(d) \vee \underline{\Diamond}\ end(d)))$$

We use $c \parallel d$ to represent execution sequences where c and d are concurrent.

Next, we consider a more complex relationship at a given level of nesting between an action c and the roles of its parent p that do not participate in c, i.e. that bypass c, as shown in Figure 4.4. We define $bypass(p, c)$ as the set of such roles of p. For any $i \in$

bypass(p, c), if *ex-objects*(c) $\cap$ *objects*(i) $\neq \varnothing$, i.e. if role i accesses some of c's external objects, then we have to ensure that the operations of role i on these external objects do not interfere with c. Here we define *ex_op*(i, c) as the set of such operations. (For a given operation *op* of role i, predicate *called*(*op*) is true in the state of an execution sequence where *op* is called by a thread and *return*(*op*) is true when *op* returns to its caller.)

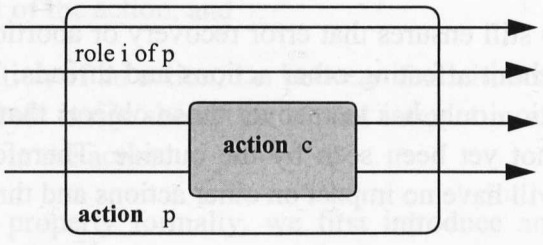


Figure 4.4 Action c and the bypass role i

In general, we use $c \parallel i$ to represent interleaved execution sequences of c and role i. The interleaved execution of c and role $i \in$ *bypass*(p, c) is said to be *proper* if action c is executed as a whole between the operations in *ex_op*(i, c) (i.e. no operation is interleaved into the execution of c), that is

$$\forall\, op \in \textit{ex-op}(i, c) \bullet \Box\ ((\textit{begin}(c) \Rightarrow \Diamond\!\!\!-\ \textit{return}(op)) \vee (\textit{end}(c) \Rightarrow \Diamond\ \textit{called}(op)))$$

We use $c \perp i$ to represent *properly* interleaved execution sequences.

Let s be a set of possible execution sequences characterized by an expression of the form $c \gg d$, $c \parallel d$, $c \perp i$ or $c \parallel i$. Let O be a set of objects. For each possible execution sequence in s, we are interested in the values of O at the end (the final state) of the appropriate action (d for $c \gg d$, c for $c \perp i$ and $c \parallel i$, and c or d, whichever ends later, for $c \parallel d$.) For any given execution sequence in s, we will have a set of end values for the objects in O. Define *end_values*(O, s) to be the set of all possible sets of end values of O for execution sequences in s.

Given these relationships defined above, we can now formalize two conditions regarding concurrent access to the external objects of an action.

1) The interleaved execution of an action c and any action $d \in C_p$ where p = *parent*(c) is in effect equivalent to one of the executions such that c indivisibly precedes d or d indivisibly precedes c, that is

$$\textit{end-values}(\textit{ex\_objects}(c), c \parallel d) \subseteq$$
$$\textit{end-values}(\textit{ex\_objects}(c), c \gg d) \cup \textit{end-values}(\textit{ex\_objects}(c), d \gg c) \qquad \text{(C1)}$$

2) The interleaved execution of an action c and a bypass role $i \in$ *bypass*(p, c) of its parent p is equivalent to one of their proper executions, that is:

$$\textit{end-values}(\textit{ex\_objects}(c), c \parallel i) \subseteq \textit{end-values}(\textit{ex\_objects}(c), c \perp i) \qquad \text{(C2)}$$

These conditions ensure that the two forms of interleaved execution have the same effects as their serial execution, that is, the consistency of shared objects is always maintained. Now, let p be the parent action of a CA action $c \in C$. For a given concurrency control protocol, if the protocol guarantees: i) for any $d \in C_p$, any interleaved execution of c and d satisfies Condition C1, and ii) for any $i \in bypass(p, c)$, any interleaved execution of c and i satisfies Condition C2, then the consistency of c's external objects is maintained. If for any $c \in C$ in a given system the above constraints are met, then the consistency of the system will be guaranteed.

4.2.5 Exceptions and Error Recovery

This subsection will briefly describe some characteristics of a CA action under an exception handling framework and provide only some high-level formalization. Exception handling and error recovery will be addressed in much more detail in Sections 4.3 and 4.4. From the fault tolerance point of view, a CA action should have a simple and deterministic behaviour, that is, either end normally or signal an appropriate exception to its enclosing action (or its surrounding environment).

Let $e(c)$ be the set of (internal) exceptions that can occur during the execution of action c, and $\varepsilon(c)$ be the set of exceptions that c can signal externally. We define two new state predicates: *raise*(e, c) is true in the state where an exception e is raised within c, and *signal*(e, c) is true in the state where action c signals an exception e to its enclosing action or its user environment.

The normal end of an action will be reached if no exception occurs or error recovery inside c is fully successful. However, the exceptional end of an action must lead to the signalling of an appropriate exception e that specifies an exceptional outcome, that is

$$\forall c \in C \bullet \Box (exceptionalend(c) \Rightarrow \exists e \in \varepsilon(c) \bullet signal(e, c)) \qquad \text{(P11)}$$

In particular, the aborted end of an action removes any effect that the action may have had on its external objects and signals a special abortion exception `abort`, that is

$$\forall c \in C \bullet \Box (abortedend(c) \Rightarrow signal(\texttt{abort}, c)) \qquad \text{P(12)}$$

The failed end of an action is reached if error recovery is not possible. In this worst case, a special failure exception `fail` is signalled, that is

$$\forall c \in C \bullet \Box (failedend(c) \Rightarrow signal(\texttt{fail}, c)) \qquad \text{P(13)}$$

Within a CA action, it is important to characterize the relative ordering of the execution states related to exceptions and exception handling. Define *handling*(e, r, c) as a state predicate that is true in the state where role r of action c starts handling the exception e.

If an exception is raised within an action, all the roles of the action must stop the normal computation and handle the exception together:

$$\forall c \in C, e \in e(c) \bullet \Box (raise(e, c) \Rightarrow \Diamond (\forall r \in roles(c) \bullet handling(e, r, c))) \quad \text{(P14)}$$

Similarly, the signalling of an exception from a nested action will cause all the roles of the enclosing action to handle the exception together:

$$\forall c \in C, e \in \varepsilon(c) \bullet \Box (signal(e, c) \Rightarrow \Diamond (\forall r \in roles(parent(c)) \bullet handling(e, r, parent(c)))) \quad \text{(P15)}$$

Finally, we discuss the effects of error recovery after exception handling has been performed by roles. Define three new predicates: *recovery*(c) is true in the state where c returns to a normal execution state, *p-recovery*(c) is true in the state where c returns to an exceptional, but valid state, and *abortion*(c) is true in the state where c returns to its initial state.

Successful error recovery leads to the normal end of an action, that is

$$\forall c \in C \bullet \Box (recovery(c) \Rightarrow \Diamond normalend(c)) \quad \text{(P16)}$$

Partial recovery can only reach the exceptional end of an action, that is

$$\forall c \in C \bullet \Box (p\text{-}recovery(c) \Rightarrow \Diamond exceptionalend(c)) \quad \text{(P17)}$$

Successful abortion ensures the aborted end of an action, that is

$$\forall c \in C \bullet \Box (abortion(c) \Rightarrow \Diamond abortedend(c)) \quad \text{(P18)}$$

And if none of the above forms of recovery was successful, the failed end of an action must be reached, that is

$$\forall c \in C \bullet \Box (\neg (\underline{\Diamond}(recovery(c) \vee p\text{-}recovery(c) \vee abortion(c))) \Rightarrow failedend(c)) \quad \text{(P19)}$$

4.3 Exception Handling in Concurrent/Distributed Systems

In this section, exception handling is considered as a general mechanism for coping with exceptional system conditions or errors caused by software faults, hardware faults, and faults that occur in the environment of the computer system but may affect both the system and its environment.

An exception handling mechanism is a programming language control structure that allows programmers to describe the replacement of the normal program execution by an exceptional execution when occurrence of an exception (i.e. inconsistency with the program specification and hence an interruption to the normal flow of control) is detected [Cristian 1995]. For any given exception handling mechanism, exception contexts are defined as regions in which the same exceptions are treated in the same way; often these contexts are blocks or procedure bodies. Each context should have a

set of associated exception handlers, one of which will be invoked when a corresponding exception is raised. There are different models for changing the control flow, but the termination model is most popular. This model assumes that when an exception is raised, the corresponding handler copes with the exception and completes the program execution. If the handler for this exception does not exist in the context or it is not able to recover the program, then the exception will be propagated. Such exception propagation often goes through a chain of procedure calls or nested blocks where the handler is successively sought in the exception context containing the context which raised or propagated the exception.

Exception handling and the provision of fault tolerance are more difficult in concurrent/distributed systems. For example, there would be no problem in sequential programs when a client object tries to get data from an empty queue — an interface exception will be signalled by the server object. However, concurrent access to server objects, permitted by concurrent systems, will complicate such exceptional situations. If two clients attempt to access a queue concurrently (when the queue contains only an element), one client may surprisingly receive an interface exception which blames it for the use of an empty queue! A more serious complication is that several exceptions can be raised concurrently in multiple concurrent activities [Campbell & Randell 1986][Romanovsky et al 1996]. Obviously, proper exception handling has to involve multiple interacting activities and additional mechanisms for coordinating multiple objects are needed.

Exception propagation in concurrent programs may not simply go through a chain of nested callers, but can require an extra dimension of propagation. In the situation of nested atomic actions, an exception may need to be propagated upward to the enclosing action from a nested action. Since the enclosing action can involve more components than the nested action, the exception may therefore also need to be propagated to all the components of the enclosing action in order to start a joint recovery activity. Unfortunately, no known language or system provides appropriate support for such two-dimensional exception propagation.

Physical distribution of computing further complicates coordination of multiple concurrent components. In a distributed system, each node may possess a separate memory; as a consequence, software segments executing on different nodes will reside in disjoint address spaces and so must communicate by the exchange of messages over relatively narrow bandwidth communication channels. The time of message passing is not negligible and the effect caused by the communication delay must be therefore taken into account.

Finally, proper handling of an exception requires an assessment of the extent of the damage it might have caused. This is difficult in systems involving complex interactions among concurrent activities. However, if exception handling is incorporated into an appropriate abstraction of cooperative activity, such abstractions can be used as a basis for damage assessment and confinement. Most of the existing

schemes for exception handling in concurrent/distributed systems use the concept of an atomic action as a unit of error confinement [Jalote & Campbell 1986][Taylor 1986], though there is no clear consensus on how to handle exceptions when asynchronous activities occur.

4.3.1 Conceptual Model

When considering the general model for concurrent/distributed system in Section 4.1, it appears a natural decision to regard CA actions as a structuring unit for performing complex exception handling in such systems. We therefore model the dynamic structure of a concurrent/distributed object system as a set of interacting CA actions. Every CA action may enclose nested actions if needed, and exceptions can be propagated over nesting levels.

Exception Declaration

For a given CA action, there are two types of exceptions: those that are totally internal to the CA action and that when raised are entirely handled within the action, and those that are known in and can be signalled to the enclosing environment (e.g. its calling thread or the enclosing action).

The set of exceptions, $e = \{e_1, e_2, e_3, ...\}$, that can be raised within a CA action must be declared as part of the action definition. The corresponding exception handlers for these exceptions are associated with the various roles of the CA action. The set of exceptions, $\varepsilon = \{\varepsilon_1, \varepsilon_2, \varepsilon_3, ...\}$, that can be signalled from a CA action to its environment should be specified in the interface to the CA action. These exceptions are signalled in order to indicate that, though internal exception handling might have been attempted (unsuccessfully), an unrecoverable exceptional condition has occurred within the action, and/or only incomplete results can be delivered by the action. For a nested CA action and its direct-enclosing action, the definitions of e and ε are fully recursive, namely,

$$\varepsilon_{nested} \subseteq e_{enclosing}$$

There are two special exceptions μ and f in ε. An `abort` exception, μ, implies that the action has been aborted and all of its effect have been undone. Since `abort` is not always possible, a `failure` exception, f, indicates that the action has been aborted but that its effect may not have been completely undone.

Exception Handling and Propagation

When a thread enters an action to play a specified role, it enters the related exception context. Some or all of the participating threads may later enter nested CA actions. Since the nesting of CA actions causes the nesting of exception contexts, each

participating thread of the nested action must be associated with an appropriate set of handlers. Exceptions can be propagated along nested exception contexts, namely the chain of nested CA actions. Three terms are used here to clarify the route of exception propagation: an exception e_i in e is *raised* by a role within a CA action, other roles of the same action are then *informed* of the exception e_i and, if handling the exception within the CA action is not fully successful, a further exception ε_j in ε will be *signalled* from a nested action to its enclosing action (see Figure 4.5).

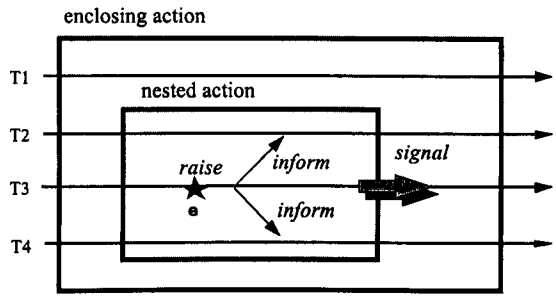


Figure 4.5 Exception propagation over nesting levels

There are at least two ways of signalling an exception from a nested action to its enclosing action. One possibility is that a "leading" role has been pre-defined by the designer, or is determined dynamically, that has the responsibility of signalling an agreed exception to the enclosing action. The other approach adopts a more distributed strategy: each role of the nested action is responsible for signalling its own exception. (Conceptually, the exceptions signalled by respective roles should be the same. It will however cause no problem for roles to signal different exceptions because an action in our model is required to have the ability of handling concurrent exceptions, the exceptions signalled concurrently from the nested action will be handled simply as if they are concurrently raised in the enclosing action.)

Distributed exception signalling requires some final-stage coordination on two special exceptions μ and f. If any role of a nested action is to signal the exception f, then all other roles of the nested action must signal the same exception f to the enclosing action, indicating that some erroneous effect made by the nested action may not have been undone completely. Similarly, it makes sense only if all roles signal the same `abort` exception, μ, in order to ensure that all the effects made by the nested CA action (more precisely, made by respective roles) have been undone completely.

Control Flow

The termination model of control flow is used here — in any exceptional situation, handlers take over the duties of participating threads in a CA action and complete the action either successfully or by signalling an exception ε_j to the enclosing action.

External Objects

Since the effect of a CA action in our system model can be observed only through the committed state of some external objects, once an exception is raised within the CA action and hence error recovery is requested, the related external objects must be treated explicitly and in a coordinated fashion, the aim being to leave them in a consistent state, if at all possible. The standard way of doing this in transactional systems is by restoring the objects to their prior states. However, an exception does not necessarily lead to restoration of all the external objects. (Indeed, external objects, particularly real ones in the computers' environment, might not be capable of state restoration.) Appropriate exception handlers may well be able to lead such objects to new valid states. But if it is detected that one or more external shared objects have failed to reach a correct state, a `failure` exception *f* must be signalled to the enclosing CA action in the hope that it may be able to handle the situation.

Exception Resolution

If several exceptions are raised at the same time, one simple method for resolving the exceptions is to prioritize them. The disadvantage of this scheme is that it does not allow representation of situations where the concurrently raised exceptions are merely manifestations of a different, more complicated, exception. To provide a more general method, an *exception graph* representing an exception hierarchy can be utilized. If several exceptions are raised concurrently, then the multiple exceptions are resolved into the exception that is the root of the smallest subtree containing all the raised exceptions [Campbell & Randell 1986]. Each CA action should have its own exception graph. For example, the graph would be specified by a keyword like **`exception hierarchy`** in the definition of a CA action, and would have a form like:

e_r: e_1, e_2, ..., e_k;
e_1: ... ... ;

where e_1, e_2, ..., e_k are the direct low-level nodes of the resolving exception e_r.

4.3.2 Dealing with Environmental Exceptions

Consider a system and its environment as two concurrent entities. We will now investigate the problem of how to deal with exceptions that occur in the environment (or environmental exceptions), but may nevertheless affect both the system and the environment.

Traditionally, identification of exceptions and the design of their handlers have been associated with later phases of the software lifecycle, such as operational specification, architectural design and implementation stages. During these later phases all the effort is made to protect the software from faults that may occur either in the environment or in the computer system itself including those faults in the design, the actual

implementation and the underlying virtual machine upon which the design executes. However, it is difficult in practice to deal with all forms of exceptions only when the development enters its later phases. Without bearing exception-related matter in mind in early phases, the information and data about certain types of exceptions, especially those exceptions that occur in the environment, are usually very limited in later stages of the development, if not completely unavailable. In fact, such crucial information and data should be collected during the early requirement phase and extracted carefully from the initial interaction with the user.

Now consider an approach that separates different concerns and introduces exception handling into different phases of the software lifecycle. Environmental exceptions must be dealt with as early as possible, starting from the requirement phase. Software design-related exceptions should be treated during the design phase, and implementation-related exceptions be addressed in the later implementation phase. During different phases, various types of exceptions must be defined carefully and the major obligation or function of their handlers be outlined. Figure 4.6 illustrates this approach based on a similar lifecycle model used in [Lyu & Avizienis 1993].

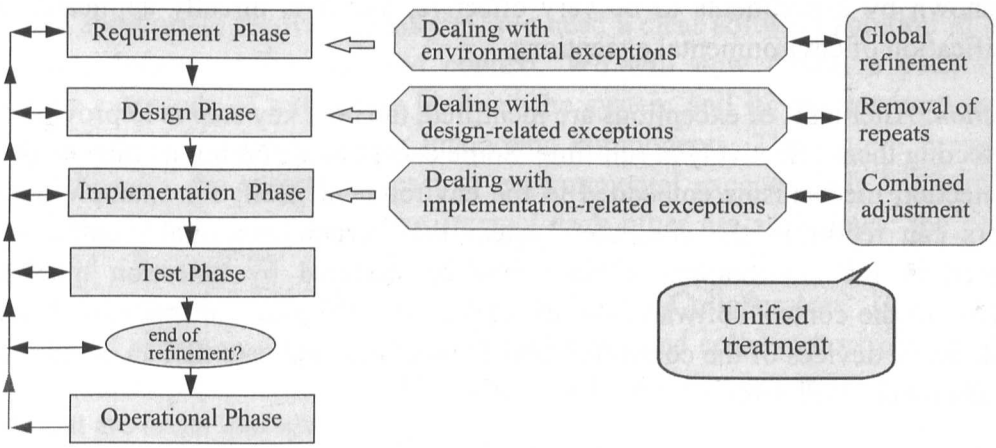


Figure 4.6 Dealing with exceptions through the software lifecycle

As the software development progresses, new exceptions are identified and their corresponding handlers are specified. Throughout the lifecycle, the process of unified treatment is essential because all exceptions identified in different phases will be ultimately handled under a unified implementation framework. This process may include a global adjustment, removal of repeated exceptions, as well as identification of certain combined exceptional conditions which could be triggered by events related to different phases. It must also consider multiple exceptions that may occur at the same time and could have a serious consequence. For example, in an auto-controlled aeroplane a failed engine, or an erroneous auto-pilot program or both will usually require completely different recovery measures.

Most existing work on exception handling is to a great extent concerned with only design-related or implementation-related exceptions [Cristian 1995]. Exceptions related

to software design are typically associated with the data structures chosen, with the means of interactions between components of the design, and with supplementary processing of the design such as fault-tolerant measures and mechanisms. Implementation-related exceptions include the invalid use of both the programming language chosen and facilities provided by the virtual machine. Run-time type errors and invalid memory access are two typical examples. Environment-related exceptions, which must be handled based on the controlled interaction between the system and its environment, have received the least attention. In the following, we will focus on issues with environmental exceptions, and will not discuss other exceptions further.

Design of Environmental Exception Handling

Identification. During the early requirement phase, the intent is to identify all exceptions that may occur in the environment of a computer system with the aid of the user. Maxion and Olszewski [Maxion & Olszewski 1998] recently developed a practical methodology for identifying exceptional conditions and improving exception handling coverage based on structured taxonomies and memory aids. Their approach was shown by experiments to be very effective, and it is directly applicable to the identification of environmental exceptions.

Detection. After a set of exceptions are identified, the next key step is to provide means of detecting them effectively at run-time. Some exceptional conditions may be detected by detection mechanisms embedded in the environment itself; for example hardware sensors can report to the computer system that certain abnormal conditions have occurred in the environment. Others may be detected by detection mechanisms involved in the control software. For example, a control program that cannot actually actuate some devices of the controlled environment may indicate that an environmental fault has manifested itself.

Handling. Defining the desired mitigating actions against exceptions is essentially a non-trivial, application-dependent activity. The functions of handlers for environmental exceptions should aim at recovering the damaged state of the environment into an acceptable known state, e.g. a pre-defined safe state.

Refinement of Handling. Mitigating actions against environmental exceptions can be further refined and detailed when the development enters the design phase. We can view the design of a software system as an algorithm which is responsible for defining interactions between components of the software, establishing connections between the software and its environment, and for providing any supplementary processing for the software to achieve its required behaviour. As the design phase progresses, it is necessary to develop a classification of anticipated triggering events and to confine the damaged scope of each event. Damage confinement and exception handling can be associated with the different levels of nested CA actions, as outlined by the conceptual model in Section 4.3.1, or alternatively can be associated with the general software architecture, i.e. overall system level, task level, and procedure/module level.

The desired mitigating actions may take such forms as substitution of default values, retry, termination of the current iteration and return, termination of the task, termination of the overall program etc. Developing a general exception handling philosophy is difficult, but some intuitive rules can be applied, including

1) At the lower level (e.g. small nested actions or procedures), the intent may be to mask the failure from the higher levels by using replication and rollback techniques; the damaged area is limited to a small scope;

2) At the higher level, the intent may be to abort and terminate a containing CA action, or a task in an orderly fashion; and

3) At the highest level (e.g. the outermost CA action or the system), an orderly shutdown of the system with an appropriate notification to the client of the system should be the primary goal of the exception handlers.

Managing Interaction between the System and Its Environment

When an exception occurs in the environment, the system is required to react properly and to take appropriate recovery measures. We need a clear software solution in terms of structure, dynamic behaviour and context. We will now devise a pattern that captures the essentials of interaction between the system and its environment, with a clear definition of interfaces among interacting components. The pattern presents a basic guideline for the system to react to environmental exceptions in a disciplined manner. Three major components that interact each other are as follows: the control software, the client of the control software, and the controlled environment with sensors and actuators. Using Class-Responsibilities-Collaborators (CRC) cards [Buschmann et al 1996], we define the responsibilities and collaborators of these three components in Figure 4.7. (Notice that only exceptions that occur in the controlled environment are taken into account.)

Class: Client	Collaborator
Responsibility • Requests services from control software • Relies on reliable and robust services	Control software

(a)

Class: Control software	Collaborator
Responsibility • Provides services to control software • Controls the controlled environment • Reacts to environmental exceptions	Client Controlled environment

(b)

Class: Controlled environment	Collaborator
Responsibility • Reacts to commands from control software • Reports exceptions to control software	Control software

(c)

Figure 4.7 Class-Responsibilities-Collaborators cards

The client requests services from the control software and relies on reliable services provided by the software (see Figure 4.7(a)). The handling of environmental exceptions and the inclusion of redundancy in the control software must be transparent to the client. Only when a request from the client is invalid or the control software is not able to provide the expected service, will an appropriate exception be signalled to the client.

In order to provide the requested services to the client, the control software interacts with the controlled environment, collecting the related data from the environment, making necessary computation, and actuating the related objects of the environment (see Figure 4.7(b)). When the environment responds wrongly or does not respond at all, the control software must invoke proper recovery activities. Also, when an exception is signalled from the environment, the software must ensure the control transfer to an appropriate handler (see Figure 4.7(c)).

In considering an object-oriented solution to the control software, several forces have to be solved: i) the software should contain a part (or a component) that is responsible for coping with environmental exceptions in which different policies and schemes can be used, ii) to control the software complexity, any exception handling part must be separated from the normal control functionalities, and iii) abstraction of handling policies must be provided to allow the customization of different policies. Figure 4.8 illustrates the structure of a possible solution.

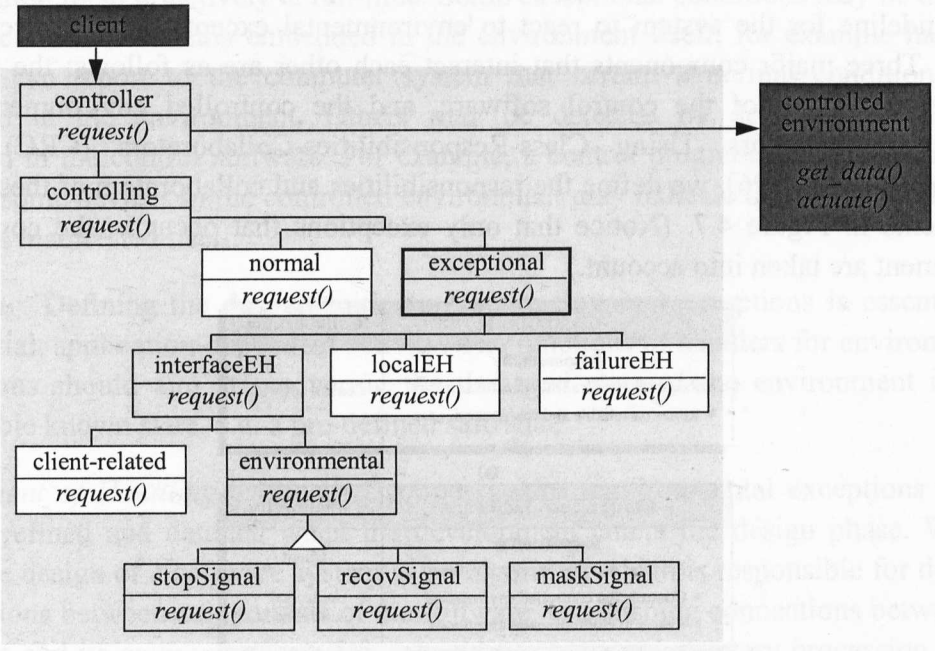


Figure 4.8 Solution structure

The main participants in the structure shown in Figure 4.8 include: `client`, `controlled environment`, `controller`, `controlling`, `normal`, `exceptional`, etc. The `client` invokes services of the `controller` when a reliable service is requested. The `controller` requests service from the `controlling` component, interacts with the `controlled environment` using the control information and results provided by the

controlling component, and returns any necessary information and results back to the client. The controlled environment class presents an abstraction of the actual environment. The controlling class has two sub-components that deal with normal and abnormal situations respectively. The normal class is responsible for normal control functionalities, while the exceptional class is responsible for handling exceptional conditions. The latter has three further sub-components. The localEH and failureEH classes are related to the handling of exceptions that occur in the computer system itself, which we will not discuss in more detail. The interfaceEH class is responsible for dealing with exceptions both in the client and in the controlled environment. To handle the environmental exceptions independently, the abstract environmental class declares the common interface for objects that implement different recovery measures. The stopSignal, recovSignal, and maskSignal components implement various recovery techniques suitable for use with the environmental interface.

Three forms of recovery techniques are used in this structure. The simplest stopSignal scheme is to simply bring the controlled environment to a previously defined state, e.g. a safe state, and to stop the entire system, sending an appropriate notification to the client. The recovSignal scheme may use forward recovery, backward recovery, or a combination of them to attempt to provide the expected services; in many cases only a degraded service can be provided with an appropriate signal to the client. The maskSignal technique is based on some redundancy in the environment to mask certain environmental faults. The expected services can be usually provided but the exceptional conditions should still be reported to the client, e.g. to facilitate the subsequent off-line repair.

The interaction diagram in Figure 4.9 shows an example of collaborations between objects in the proposed structure in the occurrence of an environmental exception. The client first invokes the services of the controller. The controller then delegates the required services to the controlling component after it gets the necessary data from the controlled environment. The controlling component computes the request and returns its results to the controller. Assuming that an environmental exception is detected during the control process, the controlling component will return the results for error recovery and exception signalling instead. The controller then actuates the controlled environment for the purpose of error recovery and sends an alarm signal to the client. (Within the controlling component, the request for exception handling is in fact delegated to the recovSignal object via components interfaceEH and environmental. It is the recovSignal component that actually computes the recovery action and prepares an appropriate notification to be sent to the client.)

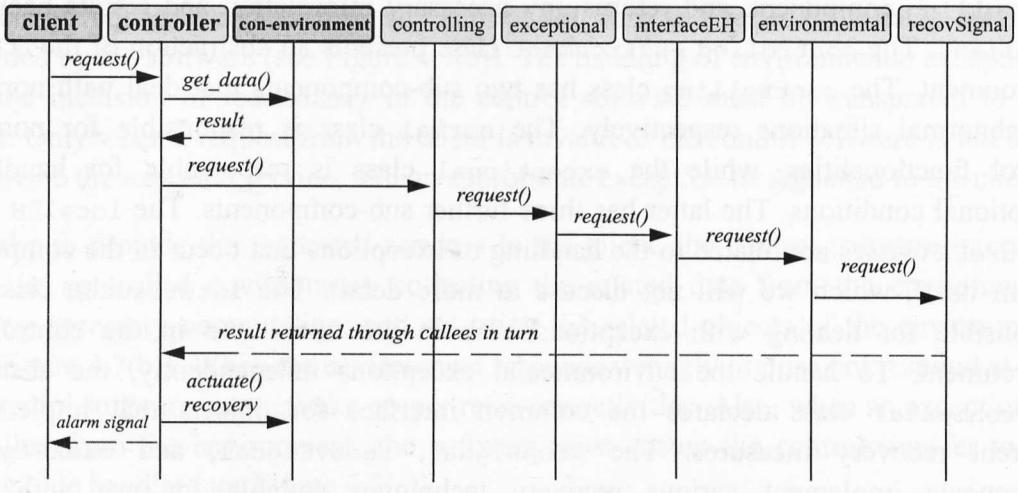


Figure 4.9 Collaborations between objects

Our pattern is supposed to be used in situations in which a system is responsible for handling exceptional conditions of its environment. The proposed structure separates different concerns and different obligations in a disciplined way, and so the complexity of the control software is well managed. In particular, the control software exercises its control on the environment through a clearly defined and narrow interface so as to minimize the negative effect of environmental exceptions on the software itself. These two important characteristics of our pattern will, we hope, lead to an effective improvement of the overall system reliability.

4.3.3 Dealing with Concurrent Exceptions

We will now study how to deal with another class of exceptions, i.e. exceptions that occur simultaneously, perhaps in different nodes of a distributed system. A detailed discussion of the necessity of coping with concurrent exceptions has been presented in [Romanovsky et al 1996]. Campbell and Randell in [Campbell & Randell 1986] argued that a hierarchy-based approach is essential for handling exceptions raised concurrently in order to find a higher-order exception that can "cover" all the concurrent exceptions. This further requires a distributed scheme for determining the proper recovery strategy and for involving all the related components in the recovery activity. In the following, we will first introduce the notion of exception graphs, then describe two algorithms for both exception resolution and signalling, and finally discuss the results of an empirical study.

Exception Graphs

The exception tree concept (first proposed in [Campbell & Randell 1986]) presents a simplified form of specifying the relationship between multiple exceptions. However we have found that in practice an exception hierarchy often has a more complicated

form than a simple tree. It is therefore important to formalize a general form of such relationship. We now define an exception graph formally below.

An exception graph is a directed graph $G(E, R)$ where the exception set $E = \{e_1, e_2, ..., e_n\}$. Each exception $e_i \in E$ is represented by a node and each directed edge $(e_i, e_j) \in R$ represents a simple relationship in which $e_i \in E$ is the direct high-level node, or parent node of $e_j \in E$. Define the in-degree of node e_i, $d_{in}(e_i)$, as $|\Gamma^{-1}(e_i)|$ and the out-degree $d_{out}(e_i)$ as $|\Gamma(e_i)|$, where $\Gamma(e_i) = \{e_j : (e_i, e_j) \in R\}$ and $\Gamma^{-1}(e_i) = \{e_j : (e_j, e_i) \in R\}$. (For example, in Figure 4.10 $d_{in}(e_1) = 2$ and $d_{out}(e_1) = 0$.)

For a given $G(E, R)$, there may exist three types of nodes. Nodes with $d_{out}(e_i) = 0$ represent primitive exceptions that cover no other exceptions. Internal nodes with $d_{in}(e_i) \neq 0$ and $d_{out}(e_i) \neq 0$ represent resolving exceptions that cover some other exceptions. The node with $d_{in}(e_i) = 0$, called the root of $G(E, R)$, represents a special *universal* exception. The raising of a universal exception usually leads to the signalling of an `abort` or `failure` exception to the enclosing action.

Figure 4.10 shows an example of a four-level exception graph containing three primitive exceptions e_1, e_2, e_3 at the level 0. The resolving exception $e_1 \wedge e_2$ at level one will be raised when e_1 and e_2 are raised concurrently. Similarly, the exception $e_1 \wedge e_2 \wedge e_3$ at level two will be raised in order to cover all the three primitive exceptions. This resolving exception may still be handled by the current exception context, or otherwise the universal exception at level three will be further raised.

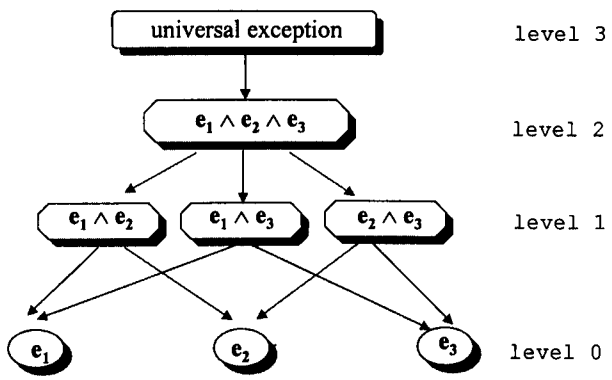


Figure 4.10 Example of a four-level exception graph

In general, an $(n+1)$-level exception graph can be defined with n primitive exceptions at level 0. The first level can contain up to $n \times (n - 1)/2$ resolving exception nodes. Level two could consist of up to $n \times (n - 1)(n - 2)/6$ nodes, and so on. Level n-1 has only one resolving exception that covers all the primitive exceptions while level n-2 may have at most n exception nodes. This general definition makes the automatic generation of an exception graph possible. However, for an actual application a simplified exception graph, though not necessarily a tree, may be required because of the considerations of space and performance. There are several ways of simplifying a general exception graph:

1) It may not be possible for certain combination of exceptions to be raised concurrently. The corresponding internal nodes (or resolving exceptions) for these combinations can thus be removed from the exception graph.

2) At a given level of the exception graph, there may exist another order relationship between exceptions, i.e. an exception may be able to cover another exception of the same level. In this case the higher order exception can be moved to a further higher level.

3) An exception graph can be structured to contain only resolving exceptions that cover certain combinations of the primitive exceptions. Other concurrently raised, primitive exceptions will simply cause the raising of the universal exception.

4) The resolving exceptions may correspond to other logical relationships. For example, the resolving exception $e_1 \vee e_2 \vee e_3$ may cover more exceptional situations than the exception $e_1 \wedge e_2 \wedge e_3$, but may require a more complicated handler.

Algorithm-Related Assumptions and Definitions

For a given CA action it is assumed that each participating thread knows the set of all other participating threads in the action and uses the same exception graph which is statically declared. Every thread has a name list for the nested actions it is to participate in. The currently innermost action for a specified thread is called the *active* CA action. Let *CA-action* be the outermost (or top-level) CA action. We define $G_{CA\text{-}action}$ as the group of participating threads $\{T_1, T_2, ..., T_i, ..., T_j, ...\}$ in *CA-action*, where each thread T_i has a unique identifier and the threads are ordered (e.g. thread names and the lexicographic ordering could be used).

During the execution of the algorithm, a participating thread T_i may be in one of the following states (denoted by $S(T_i)$):

N = Normal,

X = Exceptional (if an exception was raised in T_i), and

S = Suspended (if T_i has to stop normal computation due to exceptions raised in other threads).

Let A be the active action of T_i and G_A be the corresponding set of participating threads. We assume that each thread T_i keeps the following data structures:

list LE_i — records exceptions that have been raised, and suspended states, S, of threads that have halted normal computation;

stack SA_i — stores names of the nested actions T_i is currently in.

It is assumed that application-related message passing is treated independently, and only the following specific messages are used in our algorithm:

`Exception`(A, T_i, E) is sent by thread T_i to all the other threads of action A when an exception E is raised by T_i;

`Suspended`(A, T_i, `S`) is sent by each thread T_i that does not raise an exception but has received an `Exception` or `Suspended` message from another thread, where `S` indicates T_i is in the "Suspended" state;

`Commit`(A, E) is sent by a chosen thread in action A to all the other threads after it completes resolution of exceptions, where E is the resolving exception. A corresponding handler for E will be called by each thread once it receives this `Commit` message.

It is further assumed that an exception in an enclosing action will stop or abort any activity of its nested actions (including any nested resolution in progress and execution of any handlers).

In the interests of simplicity and brevity, our algorithm is not designed to tolerate node or communication line crashes, though a fault-tolerant version of this algorithm would be non-trivial, especially when addressing omissions and Byzantine faults. Instead, the proposed algorithm attempts to handle certain forms of software bugs, transient hardware faults and hardware design faults, but the disastrous crash of a processing node or a communication line must be masked at the appropriate underlying or hardware level, e.g. by using modular redundancy. (Our model described in section 4.3.1 is however general and it is intended to cope with exceptions that may be caused by various types of faults.)

Algorithm for Coordination and Exception Resolution

Our algorithm assumes the existence of general support mechanisms provided by the underlying system, including FIFO message sending/receiving between threads/objects and calls to abortion handlers. In addition, "< >" indicates a data item with one or more elements, "A*" is the active action of thread T_j, "$\rightarrow$" stands for "put in", and "$\Rightarrow$" stands for "sent to" in the description of our algorithm.

Figure 4.11 illustrates how the algorithm works when two exceptions `E1` and `E2` are raised concurrently in several nested CA actions. The proposed algorithm first informs all four participating threads of the two exceptions by message passing between those threads. Secondly, the algorithm aborts two nested actions because of exception `E1` that occurred outside the nested actions. (During the abortion, a further exception `E3` is signalled to the outermost enclosing action.) Finally, the algorithm determines a resolving exception `E` that covers both `E1` and `E3` and starts the handlers for `E`.

Algorithm 4.1:

For any T_i, $S(T_i)$: = N; and empty LE_i, SA_i;
loop
if T_i enters A **then**
  $<A> \rightarrow SA_i$; consume messages having arrived;
end if;
if T_i completes A **then**
  delete last element in SA_i;
  $S(T_i)$: = N if end A with success or $S(T_i)$: = X if end A with failure;
  leave A (synchronously)
end if;
if E_i is raised in T_i **then**
  $S(T_i)$: = X; $<A, T_i, E_i> \rightarrow LE_i$;
  Exception(A, T_i, E_i) $\Rightarrow$ all T_j in G_A;
  inform external objects (used by T_i within A) of the exception;
end if;
if T_i receives Exception(A^*, T_j, E_j) **or** Suspended(A^*, T_j, S) **then**
  if A^* contains or equals A **then** *//<A> is the top element in SA_i*
    $<A^*, T_j, E_j>$ or $<A^*, T_j, S> \rightarrow LE_i$;
    exception information $\Rightarrow$ uninformed external objects (used by T_i within A^*);
    if A^* contains A **then**
    abort all nested actions until A^*;
    delete the elements in SA_i until $<A^*>$;
    remove all elements except $<A^*, T_j, E_j>$ or $<A^*, T_j, S>$ in LE_i;
      if E_{ab} is raised by the abortion handler **then**
      $S(T_i)$: = X; $<A^*, T_i, E_{ab}> \rightarrow LE_i$;
      Exception(A^*, T_i, E_{ab}) $\Rightarrow$ all T_j in G_{A^*};
       else $S(T_i)$: = S; $<A^*, T_i, S> \rightarrow LE_i$;
       Suspended(A^*, T_i, S) $\Rightarrow$ all T_j in G_{A^*};
      end if;
   else **if** $S(T_i)$ = N **then** *//here $A^* = A$*
      $S(T_i)$: = S; $<A^*, T_i, S> \rightarrow LE_i$;
      Suspended(A^*, T_i, S) $\Rightarrow$ all T_j in G_{A^*};
      end if;
    end if;
  else retain the Exception or Suspended message till T_i enters A^*;
  end if;
end if;
if T_i has all exceptions, or state S, of other threads within A *//<A> is the top element in SA_i*
  and T_i has the biggest identifying number among threads with the state X **then**
  resolve exceptions in LE_i; *//find E in the exception graph*
  Commit(A, E) $\Rightarrow$ all T_j in G_A;
  empty LE_i and handle E;
end if;
if T_i receives Commit(A^*, E) **then**
  if $<A^*>$ = the top element in SA_i **then** empty LE_i and handle E;
  end if;
end if;
end loop

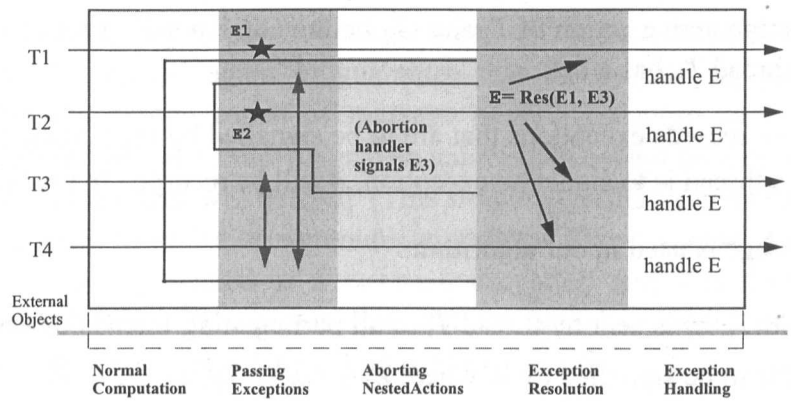


Figure 4.11 Concurrent exception handling and resolution

The proofs of the correctness of algorithm 4.1 and the analysis of its communication complexity are given in Appendixes B and C. We state two major theorems in the appendixes as follows.

> *Theorem 4.1*: Algorithm 4.1 is deadlock-free and always performs correct exception resolution.
>
> *Theorem 4.2*: In the worst case, algorithm 4.1 requires exactly $n_{max} \times (N^2 - 1)$ messages, where n_{max} is the maximum number of nesting levels of CA actions (if no nesting, then $n_{max} = 0$), and N is the number of the threads participating in the outermost CA action.

Note that the original algorithm in [Campbell & Randell 1986] is of complexity $O(n_{max} \times N^3)$. Our previous algorithm reported in [Romanovsky et al 1996] could use $n_{max} \times 3N \times (N - 1)$ messages. Algorithm 4.1 is less complex because 1) the number of messages for informing exceptions or suspended states is greatly reduced and no reply is required, and 2) only one thread (rather than all the threads) resolves multiple exceptions and only one thread needs to send the `Commit` message. In the interest of fault tolerance, the algorithm can be easily extended to the use of a group of threads that are responsible for performing resolution and producing the `Commit` messages. But this only contributes a constant factor to its total communication complexity.

Algorithm for Exception Signalling

Algorithm 4.1 ensures that a resolving exception e_r is identified and all the threads start handling this exception by invoking the appropriate handlers. However, such exception handling may be only partially successful, or fail completely. In these cases a thread must signal a further exception ε to the enclosing CA action. Following our model introduced in Section 4.3.1, participating threads of a nested action must signal the same exception μ or f if exception handling fails. We therefore need a further algorithm for coordinating those exceptions to be signalled.

Again, let A be the active action of T_i and G_A be the corresponding set of participating threads. Each thread T_i has a list:

list $listSignal_i$ — records exceptions that are to be signalled by the participating threads of action A (if a thread is to signal no exception, ϕ will be recorded in the list instead).

A specific message is used in our algorithm:

`toBeSignalled`(T_i, ε) is sent by thread T_i to all participating threads of action A when an exception ε is to be signalled by it, where $\varepsilon \in \{\phi, \varepsilon_1, \varepsilon_2, \varepsilon_3, ..., \mu, f\}$.

Algorithm 4.2:

```
//after handling the resolving exception E
For any Ti of A, empty listSignali and undo = FALSE;
loop
  if Ti is to signal ε then
    <Ti, ε> → listSignali; where ε ∈ {φ, ε1, ε2, ε3, ..., μ, f}
    toBeSignalled(Ti, ε) ⇒ all Tj in GA;
  end if;
  if Ti receives toBeSignalled(Tj, ε) then
    <Tj, ε> → listSignali;
  end if;
  if |listSignali| = |GA| then        //Ti has received all exceptions of other threads to be signalled
    switch(listSignali)
          case 1: no μ or f in listSignali
                  Ti signals ε of <Ti, ε>;
          case 2: μ but no f in listSignali
                  if undo = TRUE then
                  Ti signals μ;
                  else
                  empty listSignali and undo = TRUE;
                  Ti executes appropriate undo operations;
                  Ti is ready to signal a new exception ε;
                  end if;
          case 3: f in listSignali
                  Ti signals f;
  end if;
end loop
```

The correctness of algorithm 4.2 is obvious. In the case that neither μ nor f is to be signalled by any participating thread, no coordination will be needed; each thread simply signals its own exception or signals no exception at all. If a thread is to signal the exception f, other threads just ignore their own exceptions and signal f instead. Clearly, in these simple cases just $N \times (N-1)$ messages are required where $N = |G_A|$. In the complicated case that one thread is to signal the exception μ, all the threads must execute appropriate `undo` operations to ensure the removal of previous effects. Because

some `undo` operations may fail, in this case f, rather than μ, must be signalled and messages must be passed again to guarantee that all threads signal the same f. However, after the second round of message passing no more operations will be executed and all threads will simply signal an appropriate exception μ or f. In the worst case, $2N \times (N-1)$ messages will be used. (This simple algorithm can be easily extended to cope with crashes of nodes or communication lines. The corrupted message or lost message can be simply treated as a `failure` exception and f is then recorded in $listSignal_i$. Therefore all the threads that run on fault-free nodes can still signal exceptions correctly to the enclosing action or the calling thread.)

Empirical Study

In order to identify and tackle some implementation and performance-related issues, a prototype that supports exception resolution and nested CA actions was implemented. Figure 4.12 shows the system architecture for our prototype implementation. For each given CA action, its roles are located on separate computing nodes. Communication and interaction between these roles are supported by a message passing subsystem (MPS). Received messages are first kept in a cyclic buffer before being consumed. A run-time system (caaRTS) that supports the execution of CA actions is established together with MPS. This support system is decentralized in the sense that every distributed node has a copy of caaRTS. Apart from basic features of CA actions such as nested entrances and synchronous exits, coordinated exception handling and resolution are provided by the support system based on our algorithms. This prototype implementation shows that the distributed design of our algorithms fits well with the actual structure of a modern distributed system, and it can be easily implemented — it actually comprises about one thousand lines of Ada code [Xu et al 1998a].

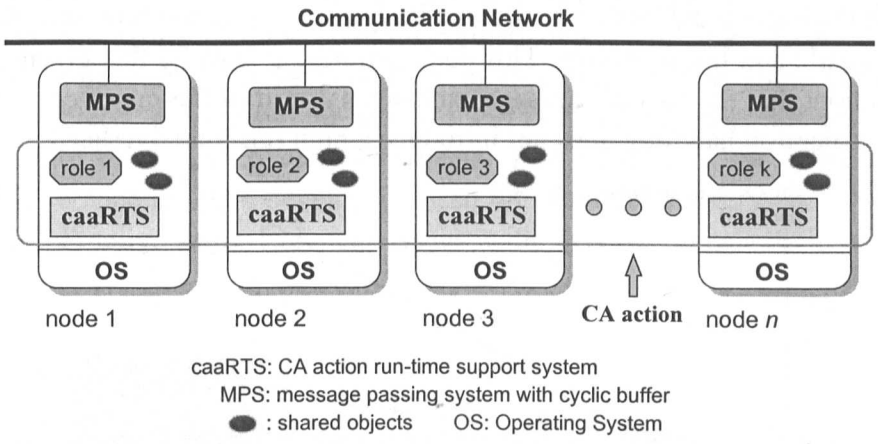


Figure 4.12 Architecture for a prototype implementation

A simple application using nested CA actions was run on the prototype implementation. The application program was executed in a 20 times loop and the execution time measured (in seconds). One of experiment sets was set up as follows:

one role of a containing action raises an exception and its nested actions have to be aborted. A further exception is raised by the abortion handler. A high-order exception (covering both exceptions) is then identified and raised in all the roles. We varied three parameters, T_{mmax}, T_{abort} and T_{reso}, in order to examine different effects of these factors on the execution time (where T_{mmax} is the maximum time of message passing between two concurrent execution threads in the system, T_{abort} is the maximum possible time for a thread to abort a nested action, and T_{reso} is the upper bound of the time spent in resolving multiple exceptions.) For example, let $T_{mmax} = 0.2s$, $T_{abort} = 0.1s$, and $T_{reso} = 0.3s$; the execution of the application will take about $94.36s$. Table 4.1 presents some typical data with varying values of T_{mmax}, T_{abort} and T_{reso}.

Message Passing	Total Execution Time	Abortion Time	Total Execution Time	Resolution Time	Total Execution Time
0.2	94.361391	0.1	94.361391	0.3	94.361391
0.4	98.586050	0.3	98.991825	0.5	98.352511
0.6	102.150904	0.5	101.939318	0.7	102.547776
0.8	106.774196	0.7	106.150075	0.9	107.164660
1.0	110.984972	0.9	110.154827	1.1	110.338507
1.2	125.078084	1.1	113.937682	1.3	114.729476
1.4	140.826807	1.3	118.147893	1.5	118.928022
1.6	161.766956	1.5	122.573297	1.7	122.483917
1.8	188.284787	1.7	128.461646	1.9	127.117187
2.0	214.519403	1.9	130.362452	2.1	131.816326
2.2	226.543372	2.1	134.165025	2.3	135.123453

Table 4.1 Performance-related results

The experimental data obtained are essentially consistent with the theoretical analysis presented previously and in the Appendixes. Figure 4.13 illustrates effect on the total execution time of the application system. When T_{mmax} is limited within $1.0s$, the cost of message passing has a minor impact on the total execution time. The execution time will increase dramatically once the time of message passing becomes longer than one second. On the other hand, with an increase in T_{reso} or T_{abort}, the total execution time has a very gentle and linear change. This demonstrates, at least by this given prototype implementation, that the cost of message exchanges is still of the major concern, while concurrent exception handling does not introduce a high run-time overhead.

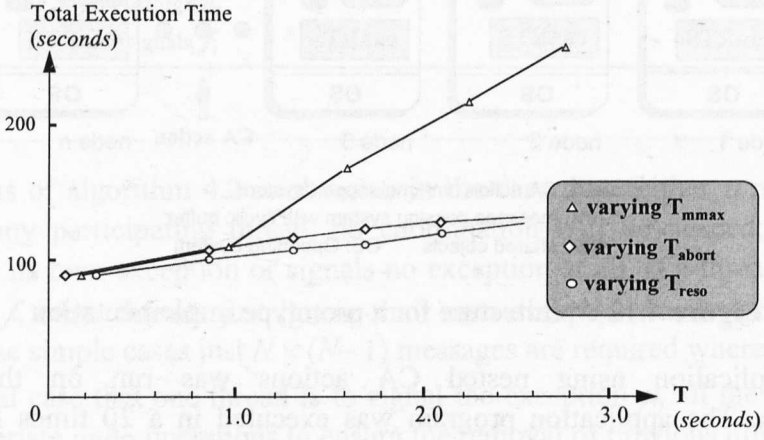


Figure 4.13 Effect on total execution time

Another set of our experiments is to compare our algorithm with the original CR algorithm developed in [Campbell & Randell 1986]. The CR algorithm was implemented by modifying our algorithm appropriately, and the same application program was used to collect the related data. The total execution time was then measured with respect to different T_{reso} and T_{mmax}. Figure 4.14 demonstrates the major change of the total execution time when varying T_{mmax} with $T_{reso} = 0.3s$, and when varying T_{reso} with $T_{mmax} = 1.0s$. A big difference in the execution time can be observed even with a fixed N ($N = 3$ in our case). In particular, the difference becomes more obvious with an increase of the time of message passing (see Figure 4.14(a)). The procedure for exception resolution is called $N \times (N - 1) \times (N - 2)$ times in CR algorithm but only once in our algorithm. This can be clearly observed in Figure 4.14(b).

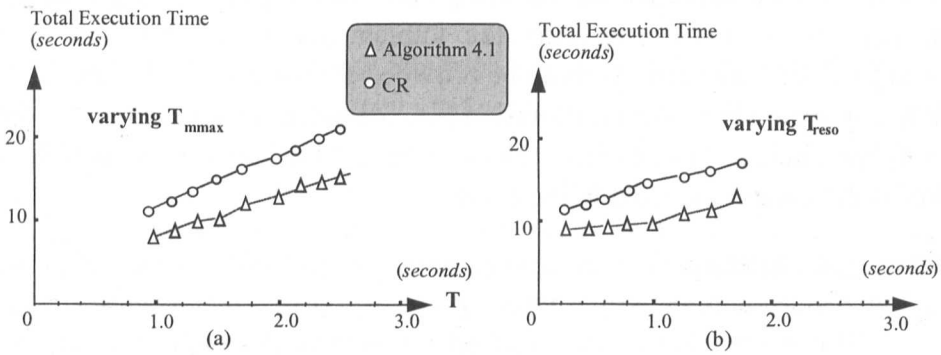


Figure 4.14 Performance-related comparison of our algorithm and the CR algorithm

4.4 Tolerating Software Faults by Design Diversity

The previous section focused mainly on issues related to the change of control flow in the occurrence of exceptions. This section will further investigate how actually to tolerate software faults in concurrent/distributed object systems. Recall the abstract system model developed at the beginning of this chapter, that contains three basic elements — objects, actions and threads. We will first address problems and difficulties associated with the design of diverse objects. Two techniques for achieving action-level fault tolerance are then developed based on adaptive recovery and fault masking respectively. Finally, some technical issues with diverse threads will be discussed briefly.

4.4.1 Object Diversity

Objects have a well-defined external interface that provides operations to manipulate an encapsulated internal state. To some extent design diversity would be well supported — different implementations can be provided for the same interface and combined together to tolerate possible design faults. However, after a close examination we have found that it is in fact not at all easy to introduce redundancy into the design of objects. Many new problems arise.

Granularity

Design diversity can be incorporated into the object model at three different levels of granularity at least: i) individual operations, i.e. methods, or part of an operation, ii) different objects from the same class, and 3) different objects from different classes. (A further level can be the meta level in the context of a reflective object system, which will be discussed in Chapter Five.)

Operation-level. The variants of an operation or part of such an operation are independently developed from the same specification. In some sense, this strategy is not truly object-oriented. However, existing techniques and experience in conventional programming can be employed most directly. For example, error recovery can be naturally achieved by restoring all modified non-local variables. In practice, further decisions need to be made regarding the implementation of this kind operation. Different degrees of transparency could be provided for the user of the special class — from full transparency as a normal operation (i.e. redundant realization of the operation is hidden by the interface) to explicit declaration where the variants, the adjudicator and the controller are clearly attached to the class.

Object-level. Fault tolerance can be also achieved by diversity in the data space of a program. For certain applications a minor perturbation of input values, or execution conditions, will often not have a major effect on outputs. A design fault in operations or computations may manifest itself under certain special data, but a set of slightly different data would cause the same operation to produce a correct output. Thus, such fault tolerance can be obtained by creating a group of objects from a class, with diversity in their internal data, and by invoking the same operation on the object group. An acceptance test is then applied to the results produced by the operation. A result passing the acceptance test, if it exists, can be used as the satisfactory output. A pilot study by Ammann and Knight [Ammann & Knight 1988] showed data diversity can be effective and very economical. Of course, redundancy at object-level could be properly combined with the operation-level redundancy.

Class-level. Redundancy at class-level is usually considered as being truly object-oriented because both the internal state and the set of operations can be independently designed from the same specification to a given type. We will mainly consider such a form of redundancy in this section. There are two similar approaches to introducing redundancy into the class-level: a set of software variants can be organized into different subclasses of an abstract class which may contain some basic information as to the specification (our C++ implementation described in Section 3.4.1 employs this approach), or the variants can be declared as different classes and regarded as different implementations to a given type (see for example the approach used in the Arche system [Benveniste & Issarny 1992] in which classes and types are treated differently). Although class-level redundancy seems to be the best choice, further problems arise, especially regarding state saving and restoration.

Diverse Objects and State Restoration

In the masking redundancy schemes, such as NVP, all variants are normally executed when invoked. Each variant can retain data between calls and, therefore, can be designed naturally as an object which hides its internal state and structure. This improves the design independence of variants and reduces the amount of data which must be passed to a variant upon invocation. In other words, diverse objects fit well with a scheme that provides fault masking. However, the adaptive redundancy schemes such as RB are different, in that they do not execute all the variants each time unless it proves necessary. Therefore all the variants must not retain data locally between calls since they could become inconsistent with each other due to the different histories of execution. If they did retain data between calls, there would be a large amount of data which must be passed to an alternate (i.e. a backup) upon recovery.

There are several ways of resolving this problem. The variants in an adaptive scheme can be designed i) as history-less functional components rather than objects, i.e. diverse design is limited to the operation-level; ii) as special objects which do not retain local data or only retain a limited amount of local data, or iii) as normal objects, but supported by distributed (parallel) execution of the variants, such as Kim's and Hecht's experiments on distributed recovery blocks [Kim 1984][Hecht et al 1989] — each variant is executed in parallel whenever invoked. In the last case, the low overhead advantages of adaptive redundancy are most lost.

Further problems will emerge if software variants are designed as objects and they retain data. For example, the system will not be able to reuse a variant which has produced an incorrect output because its internal state might have become inconsistent with the other variants. A method of dealing with this could be just to "shut down" the faulty variant. But, it is in fact critical to have a recovery mechanism that is able to recover these variants as they fail. Otherwise, the accumulation of failed variants will eventually exceed the fault-masking ability, and the entire fault-tolerant system will fail.

Recovery from transient faults requires that the state of the failed variant be brought into conformance with those forming the majority adjudication. One method of conducting recovery would be for each diverse object to roll itself back to the state that it was in prior to its last operation, i.e. to produce no result. Since each object must roll back, some of the previous history of the system will be lost. This may not be acceptable for certain applications. Another more complex method is to recover the internal state of the failed object to one which corresponds to those of the other up-to-date variants. This can easily be done if the diverse objects have the same internal data structure, such as the community error recovery method introduced in [Tso & Avizienis 1987]. Recovery is more difficult while supporting true design diversity — diverse objects should be independently designed and their internal data structures will, in general, be different. The mapping relationship between the internal state of one variant object and that of another is the key and must be obtained. If the diagnosis and

recovery mechanism judges the failure of a variant to be permanent and if an additional spare variant exists, the new object must be brought on line with its state consistent with the majority. Even in the absence of failures, we have to deal with the diverse internal state of variant objects, i.e. the problem of *replica determinism*, and guarantee that all the variants show consistent behaviour. An adjudication mechanism must decide whether the states of two or more diverse objects are equivalent based on a pre-defined equivalence relation for inexact voting. This equivalence relation should ensure the required fault coverage and at the same time decrease the probability that the adjudication mechanism fails to reach a consensus.

Tso and Avizienis [Tso & Avizienis 1987] argued the constraints on full design diversity that result from requiring the same or similar internal data structures in the variants for adjudication and easy recovery is the price paid for the increased reliability made possible by exploiting error recovery. In practical design of a fault-tolerant software system, we have to make an appropriate trade-off between the two conflict aspects: the design of internal data structures, which should support a clear and simple mapping relationship, and truly independent design of objects.

Additional Components

Given a set of diverse objects, we need some additional components to form a complete mechanism for tolerating software design faults in the objects. An adjudicator, e.g. a voter, is necessary for identifying faulty objects or ensuring that the erroneous state of faulty objects is masked and eliminated. Such an adjudication function can in some cases be provided by the underlying support system if simple (and exact) voting is possible, but for most realistic applications they have to be defined by the application programmer based on some application-specific information and they have to handle such problems as inexact voting and timing errors.

To operate on a set of diverse objects, a controller is required to actually invoke a set of operations of the objects in some form specified by the application programmer. As discussed previously, an adaptive control algorithm may be used, i.e. every time only the operation of a "master" object is actually executed. If the state of the master object fails to pass the adjudicator after the execution, a backup object will be used instead. This may however require a large amount of data passing to establish a consistent initial state for the backup object. Alternatively, the controller may invoke all the operations of the diverse objects every time in order to achieve effective fault masking. The return values, if any, are delivered only after they have been selected by the adjudication function. Other control modes and algorithms are also possible, for example based on the advanced schemes discussed in Chapter Two and Chapter Three.

4.4.2 Adaptive Recovery and Fault Masking: Two Schemes

Most existing schemes for achieving software fault tolerance in concurrent/distributed systems are process-oriented and based on the original idea behind the conversation

concept [Randell 1975], i.e. adaptive and coordinated use of a group of recovery blocks. This idea can be applied relatively easily to the structure of nested CA actions. However, to the best of our knowledge, very little work has been done on the application of masking-type schemes (e.g. NVP) to concurrent/distributed systems. With our particular emphasis on object systems, we discuss two possible schemes for incorporating design diversity into CA actions. In the interests of simplicity and brevity, we assume that no design diversity is involved in external objects, though all the techniques for creating diverse objects discussed in the last section can be combined with the two schemes to maximize the power of software fault tolerance.

Scheme One: Adaptive Recovery

For a given CA action *A* and its specification *Sp*, we will address a structural approach to enabling *A* to tolerate software faults involved in its design. An obvious requirement is that any provisions within a CA action for tolerating design faults, such as the use of design diversity, should not be visible from outside the action. We therefore refer to action *A* as the *container* action. Within the container, there is a set of nested CA actions which are designed independently from the same specification *Sp*. These nested actions are software variants. They provide the actual functionality of *A* and supply redundancy for coping with software faults.

Action Variants. Each action variant may be designed in a simplified way that permits the precise semantics of "all-or-nothing". At the beginning of a variant, recovery points as to the initial states of external objects of action *A* are established. The participating threads then play their corresponding roles within the action variant. Success of the action is the case that all participating threads leave the action and proceed, discarding the recovery points. Abortion of the action is the case that states of the external objects are restored from the recovery points, and the threads proceed, signalling an `abort` exception to the container. There is no further obligation from the action variant after its success or abortion; in either case the action is complete. In order to permit a great degree of design diversity, a thread may choose to interact with an entirely different subset of the participating threads within a new action variant as long as every variant satisfies the same specification *Sp*.

Container as Controller. The container action *A* controls the adaptive execution of action variants, which in effect involves the participating threads of *A* performing a sequence of one or more nested CA actions, depending on the errors encountered. Conceptually, the container does not need to establish appropriate recovery points since the inside action variants must do so in order to guarantee the "all-or-nothing" semantics. In practice, complex object recovery mechanisms would better be implemented at the container level so as to simplify further the development of action variants. The container action must also permit more general semantics of a CA action, including the interface checks and the provision of possible exceptional outcomes.

A container action with three action variants is shown in Figure 4.15, in which action variants are executed in an adaptive manner. If action variant 1 ends successfully, container action *A* will end with a normal outcome. That is, any effects of the final set of changes that variant 1 has made to external objects will be made visible to other actions and threads outside the container action. However, if variant 1 aborts, variant 2 will be executed after the state of external objects has been restored, assuming this is possible. (Whenever the state restoration of external objects is not feasible, the exception handling scheme we developed in Section 4.3 can be used to perform certain compensatory operations.) Again, action *A* will end normally if variant 2 ends successfully, otherwise variant 3 will be invoked. Exhaustion of all three variants without success will bring control to the last ditch part after restoration of the state of the external objects. This gives action *A* the last chance to achieve its goal. Usually, in this part *A* can only deliver some form of exceptional outcome. If even an exceptional outcome is not possible, the original recovery points are used, and an `abort` exception signalled from action *A* to its surrounding environment. In the worst case that the state restoration cannot be performed completely, e.g. the recovery points have been damaged, a `failure` exception must be signalled.

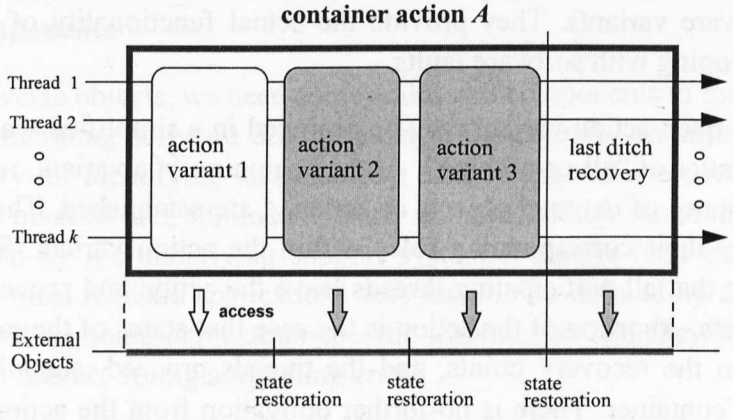


Figure 4.15 Adaptive recovery scheme

Recovery of External Objects. Because action variants are executed only sequentially in this scheme, external objects of container action *A* do not need replication, but an appropriate mechanism for checkpointing and object recovery is required. Some well-known results can be used. For example, in the Argus system [Liskov 1988] an implementation of built-in atomic objects is based on a simple locking model — read locks and write locks. Before accessing an object, an atomic action must acquire a lock in an appropriate mode. When a write lock of an object is obtained, a clone (copy) of the object is made, and the action then operates on the clone, rather than the original object. If the action commits successfully, this clone will be retained, discarding the original object. If the action aborts, the clone is discarded and the unchanged, original object is retained.

Scheme Two: Fault Masking

Though some obvious advantages are offered by adaptive schemes, for certain time-critical applications the worst-case run-time overhead imposed by such schemes is often unacceptable. A scheme that is essentially based on the concurrent execution of action variants in order to provide fault masking, may offer an effective alternative. However, the implementation of such a scheme is not quite so straightforward, because of the complications that arise through parallel accesses to objects that are external to the action (which may or may not themselves have been similarly replicated). These accesses will have to be synchronized and voted upon, as well as being subject to the normal transactional controls that are required for the use of objects that are external to a CA action. Figure 4.16 illustrates a scheme based on fault masking.

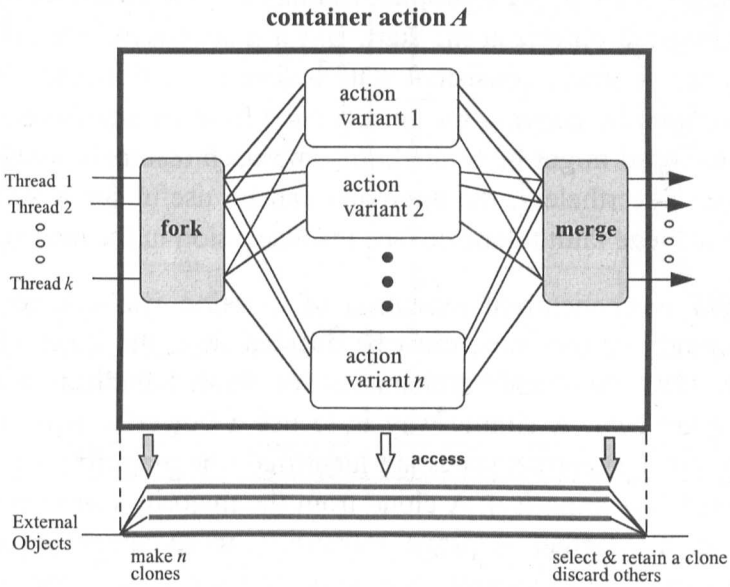


Figure 4.16 Fault masking scheme

Container as Controller. In order to exercise control, the container action A needs two additional nested actions: one forks the participating threads initially and the other merges them later, though such functionality may be provided by the system as standard procedures. As portrayed in Figure 4.16, every thread is forked into n sub-threads which will participate in n action variants respectively. The fork action is also responsible for cloning the external objects if the need arises. For example, when a write lock of an object is required by container action A, n clones of the object must be made so that n action variants can operate on these clones respectively. When all the action variants are complete, the merge action becomes responsible for selecting a correct clone for a given external object by executing an adjudication algorithm. The selected clone will be retained when the other clones and the original object are discarded. The write lock is released and the container action ends with a normal outcome. Whenever the container action aborts, for example when the merge action fails to identify a correct clone, all the clones must be discarded and only the

unchanged, original object retained. (Note that the process of making and discarding clones is equivalent to the establishment and removal of required recovery points. The nested action variants can be thus designed in a further simplified form — without consultation for the state of external objects.)

Making Clones. In the above scheme, all the clones are hidden from the world outside the container. An obvious advantage of this method is that there is no need to recover incorrect clones damaged by faulty action variants. However, since every time n clones of a given external object must be made for n action variants of the container, when objects are large, the operations of making clones can be very expensive. A possible alternative is to always keep n copies of an object during the operational time of a system, similar to some replicated object schemes for improving object availability (see [Mancini & Shrivastava 1989] for example). In this case, the container action does not need to clone its external objects at the start. But it must ensure that all object copies used will stay in the identical consistent state before the action ends. Any damaged copies must be repaired by copying the correct state from an unaffected object. While the action-level overhead might be limited, this approach results in a high systemwide overhead in space. Nevertheless, the approach can be useful for a system that uses diverse threads to achieve fault tolerance (see the discussion in the next section).

Adjudication. After the concurrent execution of n action variants, for any external object used a currently correct state must be decided from the states of its n clones. There exist many algorithms and mechanisms for result adjudication such as those addressed in Chapter Two. A simple way is to use a powerful replication checking technique like voting. The correct states are identified (the majority) and the erroneous states are eliminated (the minority). A clone from the majority can then be selected as the final outcome. To further improve reliability, more advanced and combined algorithms can be used. For example, acceptance tests can be incorporated into the adjudication process. If for any reason, no correct state can be identified, the container action will be given a final chance to decide whether to provide a degraded outcome or simply abort the entire action.

Diverse Threads

From our abstract system model, threads are the third basic element that could be replicated for the purpose of fault tolerance. However, since these replicated threads usually undertake roles of a group of action variants, they are in effect diverse. In the fault masking scheme, we have actually used the idea of diverse threads within a container action while forking a thread into n diverse sub-threads for n action variants. In principle, diverse threads can also exist outside a group of consecutive CA actions. For example, we may consider a combined use of the adaptive recovery scheme and the fault masking scheme in which forked diverse threads may need to go through a sequence of action variants as well. It is also possible to replicate threads at the level of an entire system: every thread may have n copies that are active at the same time.

Figure 4.17 illustrates two types of container actions that provide a special interface to n diverse groups of threads. There is no need for forking and merging threads, thereby simplifying the functionality of a container action. The states of external objects can be voted by a single voter (as in container action A) or using n replicated or diverse voters to further improve reliability (as in container action B). If threads are replicated at the system level, external objects of a CA action should be replicated at the same level, e.g. n copies of an object should be always maintained during the operational time of the object.

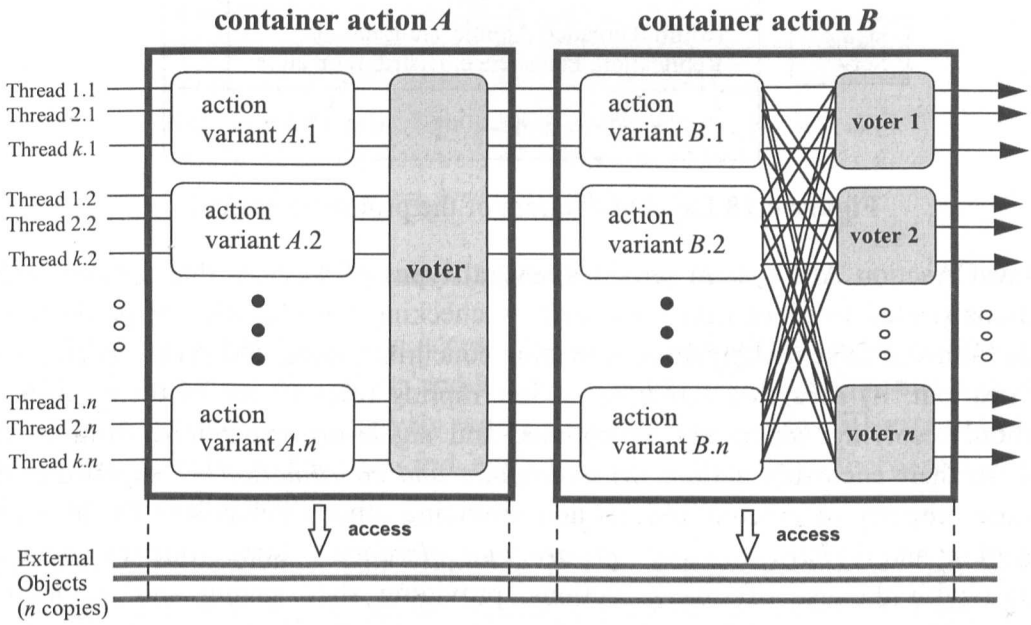


Figure 4.17 Two types of CA actions that use diverse threads

4.4.3 Prototype Implementation and Empirical Study

Since the original publication of the CA action proposal [Xu et al 1995a], we have explored various design and implementation issues for CA actions in a wide range of experiments. Among them, I have developed independently a prototype API system for programming CA actions, called JavaCAaction, which has been implemented on top of JavaArjuna. (The JavaArjuna system is an object-oriented programming system which provides a set of tools for constructing fault-tolerant distributed applications based on atomic transactions; also see Section 4.1.2). In the following, we will focus on the implementation details related to the atomicity and software fault tolerance, discussing how information smuggling [Kim 1982] can be prevented effectively and analyzing run-time overheads introduced by two CA action-based schemes for tolerating software faults.

Figure 4.18 shows a layered view of the overall system. To implement fault tolerance, applications are allowed to use both the API for CA actions and the tools provided by JavaArjuna according to different application requirements.

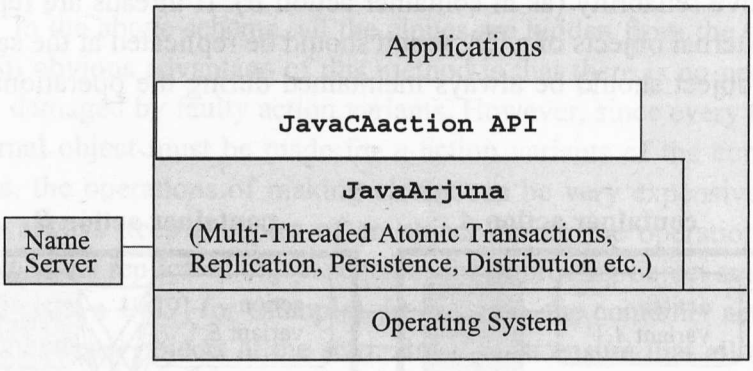


Figure 4.18 Layered diagram of the prototype system

The JavaCAaction API system provides several types of services that perform basic operations related to: i) *entrance and exit* — checking the identities of participating threads (to avoid false participants), activating concurrent roles, and synchronizing role entry and exit; ii) *exception handling* — interrupting roles if one of them raises an exception, resolving concurrent exceptions, and signalling exceptions from nested actions to their enclosing action; iii) *interaction and coordination* — supporting the enclosure property of external objects and providing other mechanisms for inter-role cooperation; and iv) *hardware and software fault tolerance* — supporting the container structure with the use of diverse actions, providing state saving and restoration, supporting the cloning and destruction of objects, and controlling the execution of the adjudicator such as the voter and the acceptance test.

There are several possible ways of implementing CA actions as a programming abstraction such as abstract data types, sets of procedures and abstract classes. However, in each case our aim is to achieve a good separation of concerns between application programmers who use CA actions, programmers who implement CA actions, and programmers who are responsible for implementing the CA action support and control mechanism (see a more detailed discussion on the separation of different concerns in Chapter Five).

In the JavaCAaction syetem CA actions are specified as normal Java classes; each CA action class provides certain managing and control functions and all the roles of the CA action are presented as associated operations. To write an actual CA action for a given fault-tolerant application, a pre-defined CA action class can be extended and can be reused by adding new action roles or by overriding the old ones. The code of an abstract `CAaction` class defined in JavaCAaction is shown below (in a simplified form).

```
import JavaArjuna.Common.*;
import JavaArjuna.Atomic.*;
class CAaction extends Thread
{
      //specify local variables and objects
      private protected Thread[] participants;
      private protected int      noParticipants;
      private protected AtomicObject[] externalObj;
      private protected Uid[]          uidExObj;
      private protected int            noExObj;
      AtomicAction A = new AtomicAction()
      private protected void entrance(Thread partThread){...}
      private protected void exit(int state){...}
      public void inAction(String newRole) throws e {...}
      public void adjudicator() throws e {...}
      public void String exceptionResolution(...){...}
}
```

Program 4.1 The `CAaction` class

The `CAaction` class is allowed to access most common tools provided by JavaArjuna, including those for atomic objects and locking policies. There are four major segments in this class, concerned respectively with i) participating threads, ii) external objects, iii) action initilization, entrance and exit, and iv) action body, the adjudicator and concurrent exception resolution.

Participating threads that call roles of a CA action are treated carefully by JavaCAaction in order to prevent information smuggling and avoid role misuse. When a thread attempts to participate in a CA action, an additional parameter is used as a `PASS` to identify the thread itself so as to guarantee that roles can be called only by the intended threads. (An alternative approach would be to call each role indirectly via a proxy object that enforced an access control policy.)

External objects in JavaCAaction must be specified as standard atomic objects supported by JavaArjuna to ensure that only consistent state transformations occur on objects despite concurrent access and failures. A simple locking policy is implemented to permit the unidirectional enclosure condition derived in Section 4.2.4, and thus recovery or abortion of any CA action can be performed alone without affecting other actions and threads. In order to provide support for software fault tolerance, the recovery strategy built in JavaArjuna is used for the AR scheme: a Java class `recoveryRecord` is defined to contain the previous state of an atomic object. The object will be brought back to the previous state in case of rollback. A deferred-update strategy used in the Argus system [Liskov 1988] is exploited for the FM scheme: invocations are performed on copies of an atomic object, rather than on the object itself.

The JavaCAaction system issues a corresponding transaction on the external objects (e.g. `AtomicAction A` in the `CAaction` class) whenever a new CA action starts, and the transaction ends when the CA action is complete. The `entrance` and `exit` operations implement the standard interface to a CA action based on the multi-threaded mechanism developed in JavaArjuna: the thread that first arrives at the CA action interface starts a transaction, and other subsequent threads will then join the same transaction, instead of starting any new transaction. The CA action can begin only if all the expected threads have arrived and all the threads have passed the identity check. At the exit, the transaction is terminated by removing all the participating threads from the transaction synchronously.

The `inAction` operation of class `CAaction` is an abstract operation which can be overridden by the programmer to write the actual code for a given action. This operation can be also re-defined as a container action to implement software fault tolerance, together with a set of diversely designed nested actions. In addition, the `adjudicator` and `exceptionResolution` operations can be overridden to realize the application-specific adjudicator and user-defined exception resolution functions.

Using JavaCAaction, we have performed several experiments to evaluate run-time overheads imposed by two CA action-based schemes for tolerating software faults. Since overheads caused by the execution mode of multiple software variants (e.g. executed adaptively or in parallel) and the extra cost introduced by various adjudicators had been well addressed by several previous experiments, we decided to focus on the issues with state restoration and object cloning, and investigate how they influence the run-time performance of CA actions.

In JavaCAaction, without using software variants a CA action that modifies an object of size 1K takes about 85 ~ 100 milliseconds to commit, including the cost of necessary disk accesses. To examine the adaptive recovery scheme, we added two nested actions into the CA action, and used two different methods to save the object state. The first method makes a clone of the object before the execution of the first nested action begins (i.e. before the execution of the primary variant), and discards the clone when the nested action ends successfully. Such `clone` and `discard` operations are quite straightforward and take less than 10 milliseconds in total. The second method makes no clone at the start, and only if the primary variant fails, the previously committed state of the object is restored from the disk, which takes about 10 milliseconds.

The fault masking scheme introduces a higher run-time cost because of the distribution of objects over a set of distributed processing nodes. For the container CA action, three nested actions corresponding to software variants were executed in parallel on three different SUN workstations. Making two clones of the original object in two remote nodes takes about 2 × 11 milliseconds, and fetching the final state of two clones back for voting requires about 20 milliseconds in total. Discarding two clones contributes only a very small part to the cost of committing a distributed container action which is

over 100 milliseconds. Table 4.2 gives the actual timings in milliseconds obtained from our experiments.

Scheme	Cloning Object	Recovery from Disk	Discarding Object	Collecting Final State
Adaptive Recovery with Two Variants	4.82	N/A	3.9	N/A
Adaptive Recovery with Disk Accesses	N/A	9.8	N/A	N/A
Fault Masking with Three Variants	2 × 10.9	N/A	2 × 3.9	2 × 9.82

Table 4.2 Timing results of object cloning and state restoration

The adaptive recovery scheme only imposes a small run-time overhead of saving and restoring the object states (although the total overhead may increase dramatically if the primary variant has failed). By contrast, the run-time overhead of cloning objects in the fault masking scheme is relatively high though acceptable. However, for any actual implementation of this scheme such performance penalty must be justified carefully.

4.5 Summary

Concurrent/distributed systems are very prone to errors because they are usually extremely complex. The incorporation of fault tolerance may be a practical method of improving their dependability. However, fault tolerance in such complex systems cannot be achieved merely by using sequential schemes such as RB and NVP in each separate thread. The problems are far more complicated when concurrency and distribution are taken into account.

The CA action concept (first introduced in [Xu et al 1995a]) addresses many problems relative to complex concurrent activities and provides special support for error recovery in concurrent/distributed object systems. CA actions were originally introduced as a structuring concept for complex system designs, and their semantics were described only informally (and incompletely) and using a simple sales control system as an illustrative example. In order to capture further the essence of CA actions, we have provided in this chapter a formal description of the CA action concept based on a linear-time temporal logic system. This formalization has been extremely helpful in clarifying a number of confusing aspects and in identifying many important properties of CA actions. In particular, we have identified the unidirectional enclosure condition that can avoid cascading recovery or abortion (a form of the domino effect) but allow a greater degree of concurrency than the strict enclosure condition. The non-enclosure conditions are also developed that permit more concurrency but may require cascading recovery or abortion in some cases.

In complex concurrent/distributed systems, exceptional conditions may arise from either the system itself or its environment. An effective method of handling exceptions

can prevent such faults from causing system failure. This is very difficult in systems involving complex interactions among concurrent activities, and exception handling facilities in current languages and systems provide little help. Using CA actions as one kind of structuring unit for damage assessment and error confinement, we have developed a general conceptual model for exception handling in concurrent/distributed systems. It is the first and very significant attempt to develop a consensus on how to handle exceptions when complex and asynchronous activities occur. Based on this conceptual model, two particular types of exceptions have been investigated in detail. For exceptions that appear in the environment of a system, we have developed a set of basic guidelines i) for the designers to improve their way of identifying exceptions and designing exception handlers during the process of software development, and ii) for the system to properly react to environmental exceptions in a structured and disciplined fashion. For exceptions that occur simultaneously, we have developed two algorithms for coordinating concurrent exception handling. Theoretic analysis and proofs with an empirical study have shown that these mechanisms are superior to the known solutions and introduce only acceptable run-time overheads.

Most existing schemes for tolerating software faults in concurrent/distributed systems have been along the line of the conversation concept using a process-oriented system model, which cannot be easily applied to object systems. Issarny was first to extend the idea of conversations to concurrent object-oriented systems, but her scheme only allows a very limited form of coordination without the provision of structuring support for effective error confinement [Issarny 1993b]. In the final section of this chapter, we have discussed different ways of applying the design diversity principle to the notion of objects. Using a top-level CA action as the container and controller, we have developed two schemes for tolerating software faults in complex object systems, featured with adaptive recovery and fault masking respectively.

These new schemes provide implementable answers to two well-known difficulties of coordinated error recovery — information smuggling and the establishment of recovery points [Gregory & Knight 1989]. The information smuggling issue can be resolved nicely as long as either the enclosure condition or the unidirectional enclosure condition is always maintained by a system or an actual implementation using CA actions. The object cloning technique, associated with both schemes, offers a practical and feasible solution to recovery of external objects that may be shared with concurrent actions and threads. The prototype JavaCAaction API system has been built on top of JavaArjuna and the experimental evidence obtained from the prototype system is quite promising.

Chapter 5

System-Level Support for Implementing Fault-Tolerant Software

This chapter investigates the issues concerned with the practical development of fault-tolerant software. Fault-tolerant software usually involves the introduction of software redundancy to normal program code. The difficulty in finding a non-intrusive way of incorporating redundancy into a complex system often hinders system development and implementation. Furthermore, in many cases the redundancy to be added is application-specific, and the developer has to address both application-dependent and redundancy-related concerns. This complicates further the task of implementing and maintaining realistic fault-tolerant software.

System-level support for developing fault-tolerant software can be provided by a variety of techniques such as high-level programming interfaces, libraries of reusable components and development environments, singly or in combination. Ideally, such support should help to i) reduce repetitive development effort, ii) ease the development process by making it a standard, and as much as possible programmer-transparent, activity, and iii) offer the application developer a wide range of fault tolerance schemes that can be selected and customised according to application-specific dependability, performance, and cost requirements.

In this chapter, we demonstrate how architectural support helps to separate different concerns and gives application programmers a simple environment for developing fault-tolerant software. Apart from the use of general techniques for software architecture, we also address the specific construction of fault-tolerant software systems with defined properties. In particular, we investigate how patterns for software architecture provide appropriate support for implementing fault-tolerant software. Finally, an industrial safety-critical application is used as a realistic case study to examine and test a variety of ideas and issues addressed throughout the whole thesis, especially those related to the actual development of complex concurrent fault-tolerant software.

5.1 Architectural Support

In order to ease the task of constructing fault-tolerant software and control its complexity, we believe, it is particularly crucial to separate different concerns properly. For the development of fault-tolerant software, the respective responsibilities of different types of programmers should be specified clearly:

> *Users of Fault-Tolerant Components (or FTC Users).* They are responsible for developing their own programs, but may use services from fault-tolerant components provided by other programmers. In general, they need to know little about how fault tolerance is achieved and implemented though they might be aware of the difference between fault-tolerant components and non-fault-tolerant components in interface and performance aspects (e.g. response time).
>
> *Programmers of Fault-Tolerant Components (or FTC Programmers).* They are responsible for developing components that tolerate certain sets of software faults. They have to address both functional requirements and fault tolerance aspects. In particular, they must be familiar with different software fault tolerance schemes and be able to develop diverse software variants and application-specific adjudicators. They are also required to select an appropriate scheme such as RB or NVP according to application-specific requirements.
>
> *Programmers of Reusable Control Components (or RCC Programmers).* They are responsible for providing the FTC programmers with a high-level and simple programming interface for the use of various software fault tolerance schemes. They are also responsible for dealing with low-level implementation details such as the schedule and execution of software variants, data consistency, state restoration, and result adjudication. Because the control part of a given scheme is often application-independent, it should be made generally reusable in order to reduce repetitive development effort.

We are interested in the development of fault-tolerant software and therefore concerned with responsibilities of both the FTC and the RCC programmers. We will study and develop a multi-level reference architecture for structuring fault-tolerant applications so that different concerns can be addressed properly at separate levels. The major idea behind our architecture is to hide the control part and the low-level implementation of a fault tolerance scheme from the FTC programmers. This enables the FTC programmers to focus mainly on functional requirements and leave the actual implementation of various software fault tolerance schemes to the RCC programmers.

However, implementing a concrete scheme in an effective way is never an easy task. Although similar implementation issues have recurred many times in a variety of experimental studies and industrial applications, the application programmers, especially novices, still have a hard time understanding and reusing existing solutions.

In an attempt to resolve this difficulty, we will use the latest pattern technique to document existing and well-proven experience, including our own experience in implementing a generic scheme for software fault tolerance [Xu et al 1995c].

5.1.1 Fault-Tolerant Components

In Chapter Two, we defined a system as a set of components which interact under the control of a design, and viewed the components themselves as systems at a lower level of abstraction in their own right.

A component should have an interface and is an encapsulated part of a system. Components serve as the building blocks for the structure of a system. At a programming-language level, components may be represented as modules, classes, objects or a set of functions. Typical examples include MODULA-2 modules, C++ classes and C functions. The CA action definition introduced in Program 4.1 is another example of a component. A CA action component may be regarded as a generalised form of a standard function, with multiple synchronous entry points and concurrent activities enclosed.

Idealized Fault-Tolerant Components

An idealized fault-tolerant component [Anderson & Lee 1981] is a (well-defined) component which includes both normal and abnormal responses in the interface between interacting components, in a framework which could minimize the impact of fault tolerance on system complexity (see Figure 5.1). Three classes of exceptional situations are distinguished: an interface exception is signalled when the interface checks of the component determine that an invalid service request has been made to the component, implying that the part of the system that made the invalid request must deal with the exception; a local exception is raised inside the component when the component has detected an error that its own exception handlers should deal with; and a failure exception is the means by which the component notifies its caller that it has been unable to provide the service requested of it.

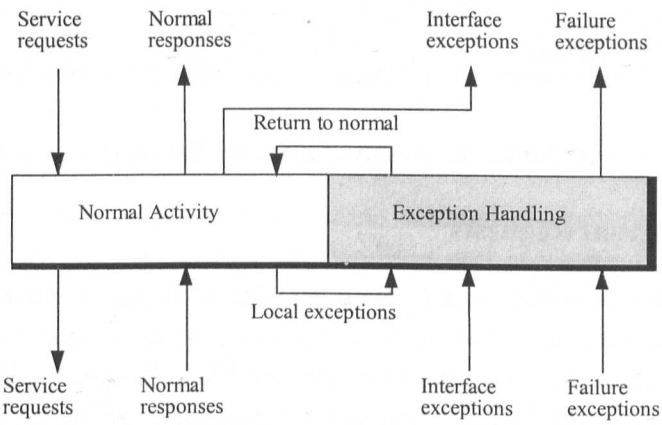


Figure 5.1 An idealized fault-tolerant component [Anderson & Lee 1981]

With such a structuring framework, it is possible, and indeed desirable, to specify the interface between each component and its environment completely. This enables the design and implementation of the component to be based on just the interface specification, and so to be undertaken independently of those of its environment, even with respect to issues of fault tolerance [Randell 1984]. However, while applying this conceptual framework to an actual implementation of a fault-tolerant component, we have to extend the framework to address three further concerns: i) degraded services, ii) concurrency, and iii) embedded software redundancy for tolerating software faults.

Extended Interface Specification

In practice, when an exception is raised within a component, in many cases the corresponding exception handler cannot deliver a complete service requested by its environment, but possibly only a degraded one. Such responses are conceptually different from both normal responses and a failure exception, and should be indicated by an attached exception. It is up to the environment to decide how to deal with a degraded service. Similarly, an abort exception should be distinguished from a failure exception. The former notifies the environment that something wrong happened inside the component but all possible effects that would affect the environment have been undone, and the latter cannot guarantee that all effects have been removed. Figure 5.2 illustrates the extended interface specification (in which each exceptional response must be indicated by the signalling of an appropriate exception).

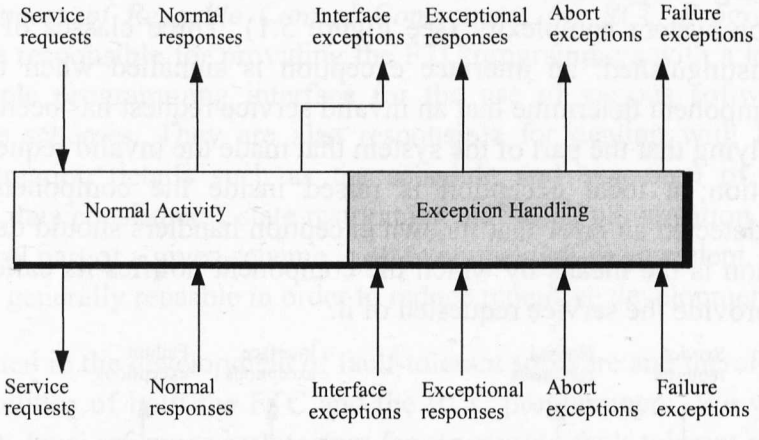


Figure 5.2 Extended interface of a fault-tolerant component

Interface for Concurrent Requests

In order to support the development of concurrent fault-tolerant software, we believe it is useful to make concurrency explicit at the interface of a component especially at a lower level of abstraction than the outermost system level. Linguistically, there are two concrete examples: the multi-function mechanism proposed in [Banâtre et al 1986] and the CA action scheme (like a multi-threaded procedure call). In general, a component may be associated with *m* synchronous entry points. Service requests may be made

concurrently through the entry points by *m* calling components or threads. In both normal and exceptional situations, the component is responsible for giving respective responses to each of the callers in a synchronous manner. Otherwise, an exception must be signalled to all the calling components. Within the component, there are exactly *m* roles that execute in parallel in response to service requests and possibly interact with each other to carry out the required computation.

Components with Diverse Design

Within an idealized fault-tolerant component [Anderson & Lee 1981], fault tolerance is obtained by a limited form of software fault tolerance — exception handling. For example, by detecting and recovering an error, and either ignoring the operation which generated it or by providing a pre-defined and heavily degraded response to that operation. In some sense, the component cannot be regarded as truly fault-tolerant since some perceived departure from specification is likely to occur. Nevertheless, the exception handling approach can result in software that is robust in the sense that catastrophic failure can be averted.

In order to incorporate (true) software fault tolerance into a system in a structured way, we need an abstraction component model that captures common characteristics of the existing software fault tolerance schemes. Figure 5.3 suggests a possible internal structure for a fault-tolerant component. There are several diversely designed sub-components called variants and an adjudicator inside the containing component. Variants deliver the same service through independent designs and implementations, and the adjudicator selects a single, presumably correct result from the results produced by variants. The containing component serves as a sort of controller that controls the execution of variants and determines the overall component output with the aid of the adjudicator.

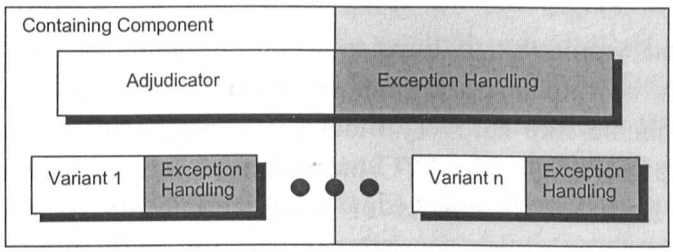


Figure 5.3 A fault-tolerant component with diverse design

Such a component should adhere to the same external characteristics as illustrated in Figure 5.2, so as to integrate exception handling and software fault tolerance into a unified framework. At run time, the containing component invokes one or more of the variants according to different schemes, and waits for the variants to finish their execution. When all variants are complete, the component invokes the adjudicator to perform a check on the results from variants and delivers an acceptable result, if one exists, to its calling environment. The whole structure is in principle fully recursive. Any sub-component such as a variant or an adjudicator can be an idealized component

as well and may have a set of exception handlers associated with it. Various software fault tolerance schemes can be implemented inside a sub-component as long as it maintains the external characteristics of an idealized component. This structure is also applicable to a component with *m* synchronous entry points. To provide the same service, each variant must have *m* roles to carry out the required concurrent computation, but may have a diverse internal design for the purpose of software fault tolerance.

The system and component structuring discussed above reflects a traditional functional view of software design. However, the structure is equally appropriate for object systems. In fact, the object-oriented paradigm fits closely with the idea of idealized fault-tolerant components. A component can conveniently be thought of as an object [Lee & Anderson 1990]. Similarly to such components, objects have a well-defined external interface that provides operations to manipulate an encapsulated internal state. Design redundancy would be well supported — different implementations can be provided for the same interface and combined together to tolerate software design faults.

5.1.2 Supporting the Development of Fault-Tolerant Components

Within a fault-tolerant component or object, apart from variants and the adjudicator, there are two other concerns related to the implementation of an actual scheme: the control structure and low-level support for the structure. For example, the recovery block scheme has the basic control structure: **ensure** `<acceptance test>` **by** `<alternate 1>` **else by** `<alternate 2>` ... **else by** `<alternate n>` **else error**, and requires a suitable mechanism for providing automatic state saving and restoration. We want to make the implementation of such control functions and low-level details transparent to the FTC programmers, thereby easing their responsibility and tasks.

The simplest method would be to develop a set of guidelines and programming conventions to show how to use a chosen language to express and implement the functionality of a scheme like recovery blocks, assuming that the language chosen provides enough expressibility. The FTC programmers must then strictly adhere to these conventions. Because all checks of adherence to the conventions can be performed only by the FTC programmers themselves, this is often a fruitful source of software bugs and may defeat the original purpose of dependability improvement.

Developing a new language that includes special features such as those related to recovery blocks would be an attractive solution. However, this could cut the work off from the mainstream of programming language developments and thus have difficulty in achieving wide acceptance. Alternatively, the pre-processor approach to extension of a popular language like Ada 95 and Java seems to be appropriate and quite practical. Unfortunately, it does have disadvantages. In particular the language provided to application programmers becomes non-standard, and programmers have in some

circumstances during program development to work in terms of the program generated by the pre-processor, rather than of the extended program that they had written.

In practice, it is always desirable that system-level support for implementing fault-tolerant software is widely available and at the same time, if possible, the development of a new language or the modification to an existing compiler is avoided. When a solution that resolves all problems is highly unlikely, it is more realistic to seek for a solution that balances well a set of contradictory problems. For example, an approach based on a library of reusable components and tools appears to be a balanced choice though it may suffer from some problems similar to the convention-based approach. This approach can free to a great extent the FTC programmers from addressing low-level concerns, thereby decreasing the complexity of implementing fault-tolerant components. Furthermore, such an approach often exploits object-oriented features, such as inheritance and polymorphism, and needs to use only a limited set of system facilities commonly found in general-purpose operating systems. Without modification of a high-level language or an operating system, rapid and instructive experiments are made possible. (The Arjuna system [Parrington et al 1995] and the ISIS system [Birman 1993] are two successful examples of taking this approach in the area of fault-tolerant distributed systems.)

Meta-level software architectures based on computational reflection have recently attracted a great deal of attention and are opening some new trends in the development of fault-tolerant distributed systems (e.g. the FRIENDS system developed at LAAS [Fabre & Pérennou 1998]). However, as a newly emerging methodology for structuring software systems, its concept and potentials have not yet been well understood. In principle, reflection is the process of reasoning about and acting upon the system itself [Maes 1987]. A reflective system can reason about, and manipulate, a representation of its own behaviour. This representation is called the system's meta level [Agha et al 1992]. Reflection improves the effectiveness of the object level or base level computation by dynamically modifying the internal organization, i.e. the meta level representation of the system. It also provides powerful expressibility and encourages modular descriptions of computation by introducing a new dimension of modularity — the separation of base level descriptions and meta level descriptions. In a reflective system a set of simple, well-defined base level features could be used to define much more complex, dynamically changeable constructs and functionalities.

Meta-level software architectures can help to separate different concerns by addressing them at separate levels. This advantage can be exploited for alleviating the complexity of developing fault-tolerant components. In fact, the structuring framework for a fault-tolerant component discussed in Section 5.1.1 facilitates the separation of base level and meta level descriptions. At a base level, simple and clear interfaces are required for specifying variants and the adjudicator and for selecting an appropriate scheme (and its control structure). Various control mechanisms that control the execution of variants can be placed at a meta level and implemented as metaobjects. The actual responses to service requests made to some base level objects is controlled and dynamically reified

at the meta level. Since a meta level object is also an object, it can be controlled by a meta meta level object. For example, the meta level operations of the control mechanisms can be further reified at a meta meta level. Low-level implementation details such as state restoration, data consistency and variant synchronization can be addressed at that level. In principle, metaobjects can be organized as an ascending tower, and hence provide multi-level modularity and support program changes in a disciplined fashion.

Given this variety of ways to support the development of fault-tolerant components, we seek for a more general solution at system-level by developing a system architecture for constructing fault-tolerant applications. We use the definition and terms of [Hayes-Roth 1995] for domain-specific architectures to describe our architectural solution. Our architecture comprises:

1) A *reference architecture* for constructing fault-tolerant applications, which consists of multiple levels corresponding to different responsibilities and concerns. For example, at the top level (i.e. the application level), there are both fault-tolerant and non-fault-tolerant components or objects, but only the fault-tolerant objects have to meet dependability-related requirements.

2) A *configuration method* for selecting and configuring components within the reference architecture to meet particular application requirements. In particular, a configuration method with simple interfaces is provided for the FTC programmers. They can customise easily a specific fault tolerance scheme based on reusable components implemented by the RCC programmers.

3) A library of *reusable components*, which contains reusable chunks of expertise in the domain of software fault tolerance. These components may be located at different levels, implementing various control mechanisms and low-level support (e.g. the state restoration mechanism). Although different components may have different responsibilities, they have to interact with each other to achieve a global goal like tolerance to software faults. A pattern is used to capture well-proven solutions and to precisely specify relationships between the components and the ways in which they collaborate.

We will discuss the reference architecture and the configuration method in the next section, and describe the pattern in detail in Section 5.2.

5.1.3 Multi-Level Reference Architecture

Figure 5.4 illustrates the static view of the reference architecture for the special application domain – software fault tolerance. This system architecture is composed of several levels: i) the *application level* for the implementation of various applications which may include a set of fault-tolerant components or objects, ii) the *system level* composed of interface components and reusable control mechanisms, iii) the *low*

system level that provides low-level mechanisms such as state restoration and variant synchronization, and iv) the *OS level* such as UNIX.

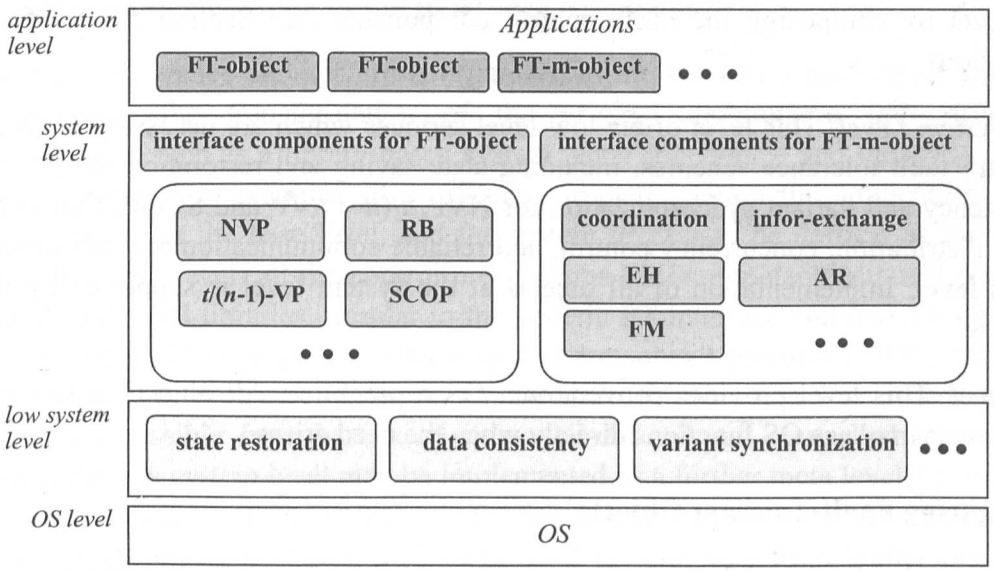


Figure 5.4 Reference architecture for fault-tolerant applications

Application Level. This level consists of application-specific objects. Prompted by dependability concerns, some critical objects may be implemented as fault-tolerant objects, and some objects may use or invoke fault-tolerant objects to perform their intended computation. Fault-tolerant objects must adhere to the standard interface characteristics of an idealized component, as shown previously in Figure 5.2. Two kinds of fault-tolerant objects are supported: standard ones for sequential invocation and extended ones for concurrent invocation via m synchronous entry points, named FT-m-object in the figure. (Section 5.2 will specify further the way of properly using reusable control mechanisms through an appropriate interface component.)

System Level. This level provides high-level support for the construction of fault-tolerant objects. There are two categories of interface components: i) external interface components and ii) generic FT interface components. The external interface components capture application-independent, external characteristics of a fault-tolerant object and specify the abstraction interface between the component and its users. The FTC programmers must follow this structuring framework (e.g. using the inheritance mechanism) when implementing a concrete fault-tolerant object.

The generic FT interface components provide the FTC programmers with a high-level programming interface that facilitates the selection and use of various software fault tolerance schemes. This category of interface components permit i) the selection of various schemes such as RB and NVP, ii) the invocation to corresponding control mechanisms, and iii) the use of application-specific adjudicators that may be "plugged in" through the interface. Pre-implemented control mechanisms include those for RB, NVP, $t/(n-1)$-VP and SCOP, and also those for concurrent programs using CA actions,

such as coordination of entry and exit, information exchange through external and local objects, exception handling (EH), fault tolerance based on adaptive recovery (AR) and fault masking (FM). More complicated control mechanisms may be implemented at this level by composing the basic control components (see Section 5.2 for further discussion).

Low System Level. This level offers low-level services which are necessary for certain software fault tolerance schemes, including state saving and restoration for RB, data consistency and variant synchronization for NVP, $t/(n-1)$-VP and SCOP. Services for object distribution, concurrency control, and reliable communication are also provided at this level. Implementation of all objects at the system level is supported by these services.

OS Level. This level provides conventional OS capabilities. All objects at the above three levels may use OS functions directly when the need arises.

Configuring Fault-Tolerant Objects

As discussed in Section 5.1, the FTC programmers must address functional aspects of a fault-tolerant object; that is, they are responsible for developing a set of software variants that satisfy the same functional requirements and for implementing an application-specific adjudicator if needed. They are also responsible for certain fault tolerance aspects of that object. More precisely, they must i) provide the fault-tolerant object with an external interface to the FTC users according to a specified structuring framework (e.g. idealized fault-tolerant components), and ii) determine a software fault tolerance scheme that controls the execution of the software variants and adjudicates the results generated by the variants.

By the aid of the reference architecture and reusable components, the FTC programmers can define a fault-tolerant object simply by:

1) reusing (e.g. through inheritance) the standard structure specified by an external interface component to implement an application-specific interface to the users of the fault-tolerant object, and

2) specifying a software fault tolerance scheme by selecting the corresponding control mechanism and plugging in the application-specific adjudicator, if needed, through a generic FT interface component.

To request services from a generic FT interface component, the FTC programmers need to pass the references of the variants and the adjudicator to the interface component. They may also have to specify the maximum number of processors required for the chosen scheme, and the objects that keep input and output data. The generic FT interface component should be an idealized fault-tolerant component, and its execution will only produce one of the following five forms of outputs: a normal

outcome, an exceptional outcome (signalling a specific exception to the fault-tolerant object), an interface exception, an abort exception, or a failure exception.

A Reflective Implementation of the Reference Architecture

The proposed reference architecture is organized in the form of a multi-level system. This facilitates the transformation of the original architecture into different implementations such as a library-based system using inheritance and delegation or a reflective system. For example, in a reflective implementation the application level may be regarded as the base level, and the system level may be defined as the meta level. At the meta level, all the control mechanisms become metaobjects and may be organized as several libraries. Similar to the generic FT interface components defined in the original multi-level architecture, a special metaobject protocol (MOP) must be implemented. This protocol serves as an interface to the meta level, and makes metaobjects accessible from base level in a well-defined and controlled manner. If necessary, the low system level may be implemented as a further meta level, or in other words, a meta meta level. In general, a reflective software system can have an infinite number of meta levels in which each meta level is controlled by a higher one, and where each meta level has its own metaobject protocol or MOP. In practice, most existing reflective languages or systems comprise only one or two meta levels. Figure 5.5 illustrates a reflective implementation in the distributed environment we used for our experiments in Section 3.4, Chapter Three.

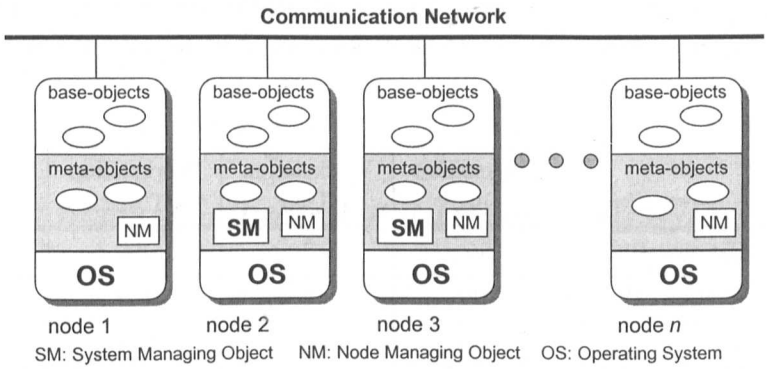


Figure 5.5 Reflective architecture in a distributed environment

Fault-tolerant objects in the original reference architecture, including software variants and the adjudicator, have been written in C++ and implemented at the base level. Various control mechanisms for different software fault tolerance schemes have been implemented at the meta level in Open C++. (Open C++ [Chiba & Masuda 1993] is a reflective version of C++ that provides the programmer with two levels of abstraction: the base level, like traditional C++ object-oriented programming; and the meta level which allows certain aspects of C++ to be redefined. In Open C++ operation calls to base level objects can be *intercepted* at the meta level by metaobjects.)

A major advantage of this reflective implementation is that for a given fault-tolerant object, the software fault tolerance scheme associated with it can be changed dynamically (even at run time) by changing and composing different control mechanisms at the meta level. When such dynamic changes are often desirable for certain applications, a reflective implementation minimizes the possible impact of the changes on application programs at the base level. However, a reflective implementation adds overheads to applications because of extra-level computation and indirection mechanisms. Its effectiveness also depends upon reflective facilities provided by a target implementation language.

We have conducted a set of experiments to find out different run-time overheads imposed by different implementation approaches. In particular, the experiment sets used in Section 3.4 have been re-run on four SUN workstations in an experimental distributed environment. Table 5.1 summarizes run-time overheads introduced by reflective operation calls. Normal C++ operation calls at the base level take from 3 to 64 microseconds, depending upon the location of different computing nodes. The corresponding operation calls in a reflective scheme (supported by Open C++) take from 6 to 100 microseconds. The overhead ratio is about 156% to 200%. However, the overhead imposed by reflective operation calls is insignificant in comparison with the entire cost introduced by fault tolerance schemes in both a library-based approach and a reflective approach. Moreover it can be seen to contribute an even smaller part to the whole overhead if the communication cost is taken into account. (These performance figures obtained from our experiments [Xu et al 1996] are very similar to the data generated subsequently from the FRIENDS system developed at LAAS [Fabre & Pérennou 1998].)

	Workstation One	Workstation Two	Workstation Three	Workstation Four
Normal operation call	56	3	64	8
Reflective operation call	100	6	100	16
Ratio	1.78	2	1.56	2

Table 5.1 Run-time overheads imposed by reflective operation calls

5.2 Architectural Pattern for Implementing Fault-Tolerant Objects

In this section we will detail further our solution to the problems that arise in designing and implementing fault-tolerant objects (as discussed in Section 5.1). Our solution scheme describes a pre-defined set of reusable components, classes or objects (located at separate levels of the reference architecture), and details their responsibilities and relationships, as well as their cooperation. The solution should be presented in an appropriate form so as to help the programmers to understand and grasp its essence. We therefore decided to use the latest pattern technique to capture the static and

dynamic structures of a fault-tolerant object, including rules and guidelines for organizing relationships between the interacting components.

Patterns also help in the system-level context of software architecture. Most patterns do not usually exist in isolation. A pattern depends upon the smaller patterns it contains and upon the larger patterns in which it is contained. A pattern-based system or pattern system [Buschmann et al 1996] with well-defined criteria for the interaction between patterns will provide a simple and natural way to incorporate software redundancy (e.g. the redundancy enclosed in a fault-tolerant object) into a large and complex system.

Design patterns can be described using different formats. The format used here is based on the work of Buschmann et al. [Buschmann et al 1996]; it contains the following parts:

- Name
- Context
- Problem
- Solution principle
- The structure and roles of classes in the solution
- The responsibilities and collaborations among classes
- Implementation guidelines
- Known uses
- Consequences
- References to related patterns

5.2.1 Pattern: Generic Software Fault Tolerance (GSFT)

Name

A pattern must be named, preferably with an intuitive name that conveys the essence of the pattern. A good pattern name is vital, as it will become part of the design vocabulary [Gamma et al 1995]. We decide to name our pattern *Generic Software Fault Tolerance (GSFT)*. This GSFT pattern provides a general way of implementing software components or objects that have the ability to tolerate residual software faults based on a variety of software fault tolerance schemes such as RB, NVP, or more advanced approaches like $t/(n-1)$-VP and SCOP. The pattern can also be used to implement the fault-tolerant objects that respond to multiple concurrent requests and enclose concurrent activities based on the CA action scheme.

Context

The GSFT pattern is intended for use when the need for ultra dependability of a (sequential or concurrent) object system arises, and the cost of developing multiple

software variants and other associated costs are justified. The system may be built from a set of interacting objects, but only typically some critical objects need to be fault-tolerant and must provide reliable services for other objects.

Problem

Software fault tolerance is often necessary, but can itself be dangerously error-prone because of the additional effort that must be involved in the programming process. The incorporation of additional redundancy can be quite intrusive and may increase overal system size and complexity, thereby adversely affecting software dependability. For example, if the application that needs fault-tolerant services has to handle fault tolerance itself, the resulting system will face several dependencies and limitations. The system becomes dependent on the software fault tolerance mechanism used, clients need to know many implementation details of the chosen fault tolerance scheme, and in many cases the solution is limited to a specific programming language.

In practice, a complex software system should be built from a set of decoupled and interacting components. This helps to control the system complexity since a component only has to address a specified concern by providing a required service. Applications that use a service should not depend upon system-specific details. For example, a client object that uses a fault-tolerant object should only see the interface offered by the fault-tolerant object. It should not need to know anything about the implementation details of fault tolerance aspects. Similarly, to achieve fault tolerance a fault-tolerant object needs to use services provided by a generic FT interface component, but should not know how a concrete fault tolerance scheme is actually implemented.

The GSFT pattern is used to balance the following forces:

- The application, or client, objects, the fault-tolerant objects, and the generic FT interface objects should interact only through well-defined interfaces. Fault-tolerant aspects should be hidden from the client objects and implementation-specific details of fault tolerance schemes should be hidden from the fault-tolerant objects.
- A generic FT interface component should provide the fault-tolerant objects with a general framework in which a concrete fault tolerance scheme can be treated as a special case and may be specified at run-time. In this way, the same design and control structure can be reused for a variety of fault tolerance schemes.
- An easy and flexible way should be provided to add (or remove) software variants and the adjudicator to (or from) the system.

Solution

Based on the multi-level reference architecture that addresses different concerns at separate levels, our pattern combines the structured characteristics of idealized fault-

tolerant components with the extensibility and flexibility offered by the object-oriented approach.

The application objects, or the clients of a fault-tolerant object, address their own functional requirements and request services from the fault-tolerant object through a well-defined interface. This interface produces one of the following five forms of responses: a normal outcome, an exceptional outcome, an interface exception, an abort exception, or a failure exception. A special base class (i.e. an external interface component) is defined for fault-tolerant objects from which an application-specific fault-tolerant object can be derived, with inherited interface methods overridden.

The fault-tolerant objects provide dependable services for the clients. They address functional aspects by implementing several software variants for the same functionality and a specific adjudication function. The fault-tolerant objects request services from a generic FT interface component and must pass the references to the variants and the adjudicator to the interface component.

A generic FT interface component actually controls the execution of the software variants and adjudicates their results. It allows fault-tolerant objects to specify a particular fault tolerance scheme and, if needed, to change the scheme to another at run-time. This interface component contains a core component called the FT controller from which various schemes can be derived and implemented. Implementing the control mechanism for a special scheme such as RB requires further low-level services including state saving and restoration, but the FCC programmers should not need to know any implementation details of those low-level services.

5.2.2 GSFT Structure

The notation used in Figure 5.6 for inheritance, delegation, aggregation and classes has the usual meaning and is the same as the notation used in [Gamma et al 1995]. In Figure 5.6, the client object invokes services of the fault-tolerant object when a fault-tolerant service is requested. The fault-tolerant object is derived from the external interface component to conform to the interface characteristics of an idealized component. It also requests services from the generic FT interface component to execute software variants and the adjudication function.

The generic FT interface component i) requests services from two or more functionally equivalent but diverse software variants, ii) sends the results of the variant executions to the adjudicator, iii) receives results back from the adjudicator and iv) reports the results back to the fault-tolerant object (which returns the results back to the client in turn). This interface component also contains an FT controller from which the RB, NVP, $t/(n-1)$-VP and SCOP subclasses etc. can be derived. These subclasses are responsible for actually controlling the execution of software variants and the result adjudication.

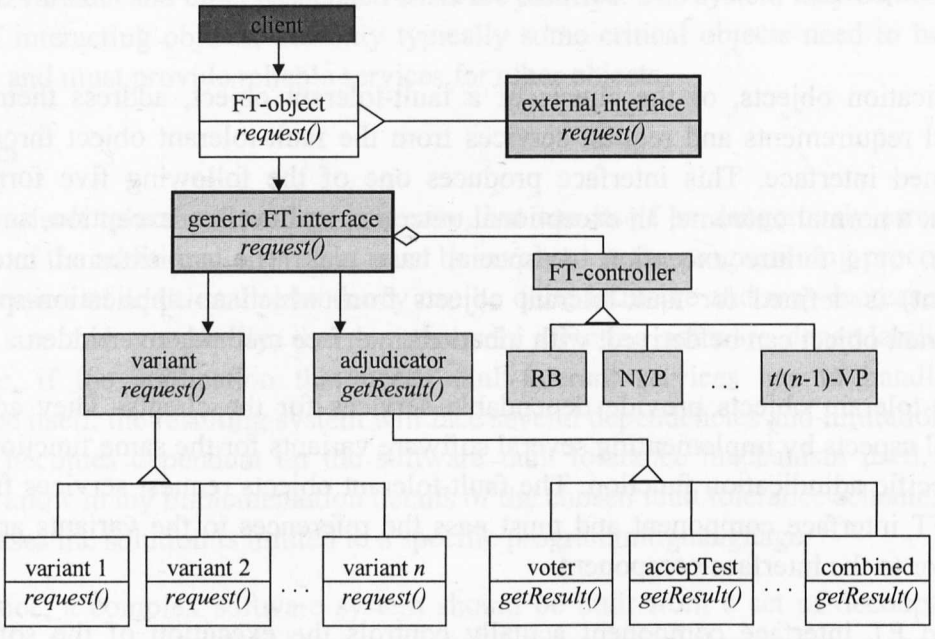


Figure 5.6 Structure of the GSFT pattern

The variant is an abstract class that declares the common interface for software variants, and *n* variant subclasses execute the requested computation and report the results back to the generic FT interface component. Similarly, the adjudicator is an abstract class that declares the common interface for adjudication functions. The voter, accepTest (i.e. the acceptance test) and combined (i.e. the combined use of voters and acceptance tests etc.) can be derived from the adjudicator class to implement actual adjudication schemes.

The structure of Figure 5.6 is also applicable to concurrent programs. Figure 5.7 shows a slightly extended structure which provides software fault tolerance based on an atomic action framework. Take the CA action scheme as a general instance. Clients 1, 2, ... and *m* are the participants of a CA action. They enter an FT-m-object synchronously by requesting services of the object. The FT-m-object behaves like a CA action and must inherit the standard CA action interface declared by the external interface component. It also requests services from the generic FT-m-interface component to control the execution of its software variants and adjudication function. The generic interface component contains an FT-m-controller from which the EH (exception handling), AR (adaptive recovery) and FM (fault masking) subclasses etc. can be derived to implement their respective schemes for tolerating software faults. These subclasses are responsible for actually controlling the execution of software variants and performing the corresponding adjudication operation on the results produced by the variants.

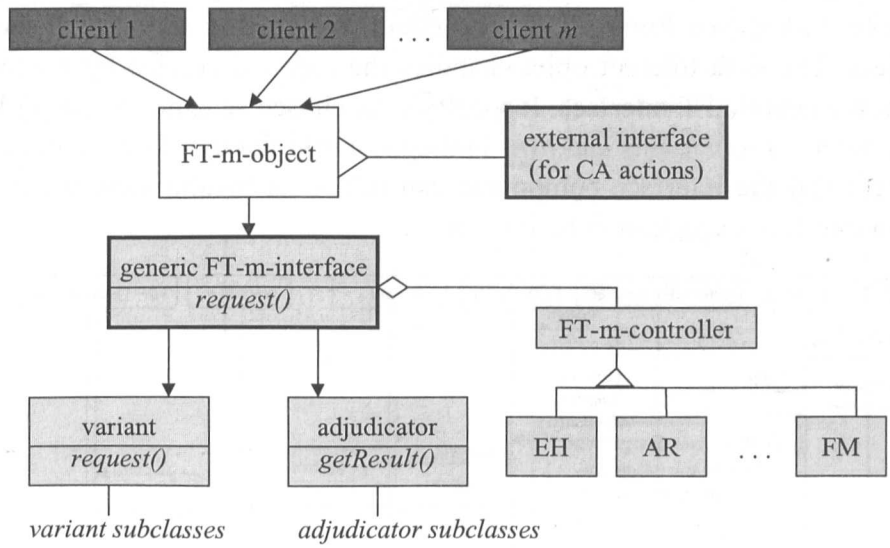


Figure 5.7 Extended structure for concurrent fault-tolerant software

The CA action scheme is developed for constructing complex concurrent systems and so is far more complex than a scheme for sequential programs. To implement the CA action scheme, the FT-m-object requires some additional support including the implementation of cooperative roles, external shared objects and local shared objects. We will discuss these problems and our solutions further when we investigate an actual application example in Section 5.3.

Finally, for a particular fault tolerance scheme, certain low-level services may be needed. Take the RB scheme as an example again. State saving and restoration are required. They can be as simple as making a copy of the original object or as complex as recovery cache memory implemented in hardware [Lee et al 1980]. However, the RB component should not need to know the low-level implementation details. It just requests a service from the state restoration component to save the system state prior to the execution of a variant and to restore the state if the execution fails to satisfy the acceptance test. Figure 5.8 shows this simple client-server structure.

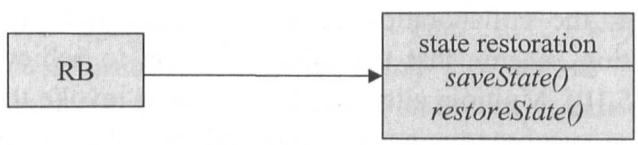


Figure 5.8 Structure for the use of low-level services

Dynamics

It is difficult to describe the dynamic behaviour of fault-tolerant software systems in general. We therefore present two typical scenarios based on the $t/(n-1)$-VP scheme for sequential programs and the CA action scheme for concurrent/distributed systems.

Scenario I illustrates the collaboration between components in the pattern that represents $t/(n-1)$-VP (see Figure 5.9). The client invokes the services of the fault-tolerant object. The fault-tolerant object handles the required service by requesting a service from the generic FT interface. It specifies the chosen scheme as $t/(n-1)$-VP and passes the references to variants and $t/(n-1)$-diagnostor to the interface. Both the fault-tolerant object and the interface component can refuse an invalid service request by signalling an interface exception to their client.

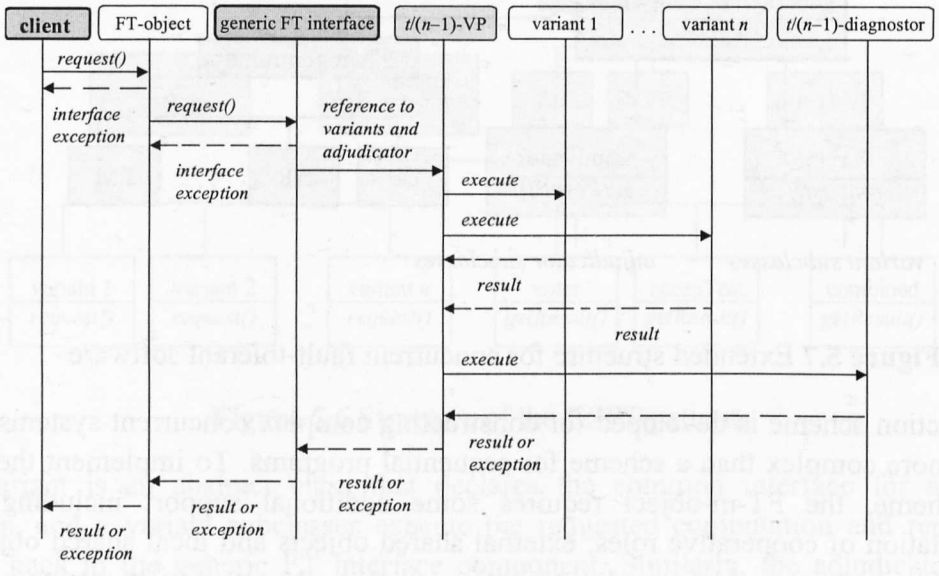


Figure 5.9 Interaction diagram for $t/(n-1)$-VP

The generic FT interface then delegates the requested service, via the $t/(n-1)$-VP controller, to n functionally equivalent software variants. Each variant performs the requested computation and returns its result to the controller. The $t/(n-1)$-VP controller then forwards the results to the $t/(n-1)$-diagnostor. Next, the diagnostor adjudicates the results and returns one of them as the correct answer, or it signals an exception. The generic FT interface receives a correct result or an exception from the controller and determines its own response to the fault-tolerant object. Finally, the fault-tolerant object returns a result back to the client or signals an appropriate exception.

Scenario II illustrates the collaboration between components in the pattern that represents the CA action scheme that uses fault masking to achieve software fault tolerance (see Figure 5.10). Multiple clients 1, 2, ..., and m invoke the services of the FT-m-object through m synchronous entry points and pass the references to external shared objects to it. The FT-m-object handles the required service by requesting a service from the generic FT-m-interface. It specifies the chosen scheme as FM (fault masking) and passes the references to external objects, variants and the voter to the interface. Again, both the FT-m-object and the interface component can refuse an invalid service request by signalling an interface exception to their clients.

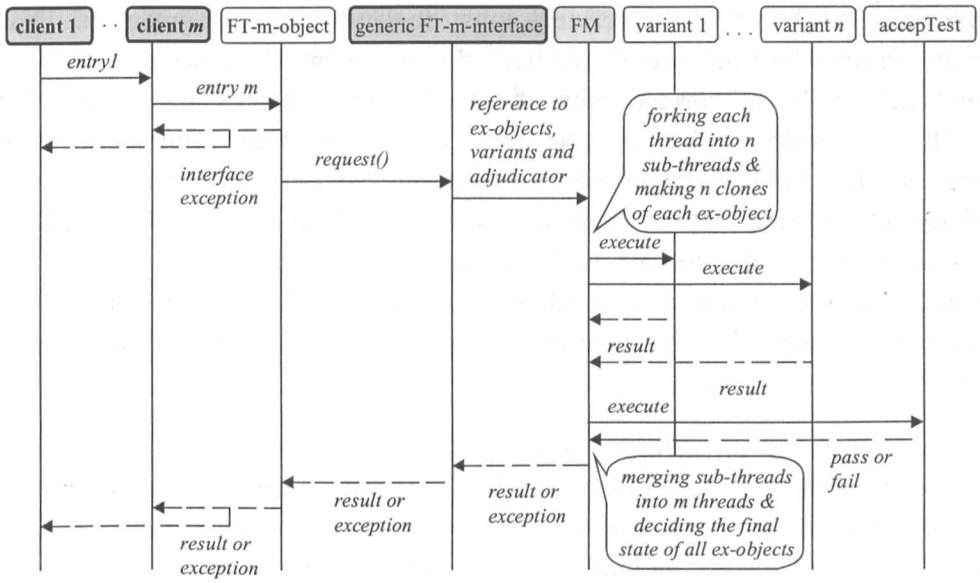


Figure 5.10 Interaction diagram for CA actions using FM

The generic FT-m-interface then delegates the requested service to the FM controller. The controller invokes the execution of n functionally equivalent software variants by forking each thread (corresponding to a client) into n sub-threads and making n clones of each external object. The m roles of each variant act upon the corresponding clones of the external objects, and all the clones are forwarded to the voter. The voter attempts to adjudicate these clones and select one of them as the correct answer or it signals an exception if no majority is found. The FM controller then merges the sub-threads into m threads and decides the final state of all external objects. The generic FT-m-interface determines its own response to the FT-m-object based on the final state of the external objects. The FT-m-object finally returns an agreed result (i.e. the current state of the external objects) back to the client or signals an appropriate exception.

5.2.3 GSFT Implementation

We have discussed many important implementation issues in the previous chapters, including software fault tolerance for sequential programs in Chapter Three and for concurrent/distributed systems in Chapter Four. We will also deal with implementation-related problems, such as using CA actions to structure realistic applications, in more detail when we investigate an industrial case study in Section 5.3. Here, we consider some of most typical issues in the implementation of the pattern, which are applicable to both sequential and concurrent software systems.

System analysis and development. First of all, use an appropriate analysis method to define a model for the given fault-tolerant application. Identify the services the software should provide, the components that fulfil these services, and the relationships and collaboration between these components. Secondly, analyze the model developed in the first step and determine which of the application services may request fault-

tolerant services and which of the services must be fault-tolerant themselves. Define fault-tolerant services that are used to address the dependability aspects of the entire application, and decide the corresponding fault tolerance schemes that support these services. Thirdly, define a set of reusable components that implement control mechanisms for the fault tolerance schemes chosen. Identify and define low-level services that support these control mechanisms. Finally, define a generic FT interface that serves as an external interface to fault-tolerant services or objects. The fault-tolerant objects can specify a particular scheme, or change from one to another, through the interface. The generic FT interface is responsible for performing all the required changes statically or at run time.

Controlling the execution of software variants: There are several key technical issues we have to deal with very carefully when implementing a software fault tolerance scheme. One of them is to control the execution of software variants. Different implementations are available, including the solutions developed in [Xu et al 1995c], [Rubira & Stroud 1994] and [Tso & Shokri 1996], with different trade-offs between the forces of simplicity, generality and flexibility. We provide an alternative object-oriented solution here based on the Composite pattern introduced in [Gamma et al 1995]. This alternative implementation suggests a neat way of using inheritance and aggregation, leading to a simple, but very flexible mechanism for controlling the execution of software variants. The control mechanism is general enough for any fault tolerance scheme using multiple variants. We take the adaptive recovery (AR) scheme for CA actions as a fairly general example to explain this implementation strategy.

Figure 5.11 shows the control structure based on the Composite pattern. The AR controls the execution of several variants by sending requests to class variant. The variant class is actually an abstract class that provides a common interface for a set of concrete variants that perform the operations requested. Apart from *n* subclasses implementing software variants, an additional subclass, called Controller, is organized as an aggregate of those variants and performs the actual control operations. The Controller has to know how many and which variants are needed for a particular request and how they are executed (e.g. sequentially, adaptively, or in parallel). To control the execution of variants, a `Controller` object has to create and store a list of concrete variant objects of its sibling classes.

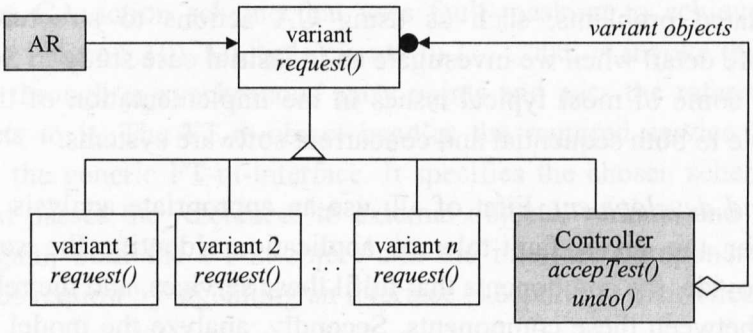


Figure 5.11 Control structure for CA actions using AR

According to the requests received, an AR object decides how many and which variants are needed and then adds the chosen variants to the Controller object. An add( ) operation must be declared in the variant abstract class in order to allow the AR object to add the variants to the Controller. This add( ) operation must be defined in the Controller subclass, but may not in the other variant subclasses if they do not support the addition operation. The sample code in C++ is as follows:

```
Class AR: public FT-m-controller {
public:
  AR( ){
      createVariant( );        //constructor for AR
  }
private:
  Variant *variant;
  void createVariant( );       //create and initialize variants
};
void AR :: createVariant( ) {
  variant = new Controller;  //create a Controller object
  variant.add(new variant1); //add all the variants chosen
  variant.add(new variant2);
  … … … …
  variant.add(new variantN);
}
```

Program 5.1 The AR class

For the AR scheme the order of addition determines the order in which variants are tried. Because the subsequent variants are executed only if their predecessors fail to satisfy the acceptance test, there is a need to maintain information about which variants have failed and which one should be tried next. This can be done by maintaining the state of the current variant being executed, e.g. using a variable called currVariant. The AR class has a private variant instance variable. The createVariant operation can initialize the variable to an instance of any variants if fault tolerance is not required (e.g. variant = new variant1;) or to an instance of the Controller class for the purpose of software fault tolerance. The Controller class may be defined as follows:

```
Class Controller: public Variant {
public:
  Controller( );
  void add(Variant*);
  void undo( );
  int action(externalObject);        //application-specific operation
private:
  int accepTest( );                  //acceptance test
  List<Variant*> variants;
  Variant* currVariant;
};
```

Program 5.2 The Controller class

The `Controller` constructor has to initialize a list of variants, i.e. the instance member `variants`, and initialize the current variant pointer: `currVariant = variants.first( )`. The `add` operation can be used by an `AR` object to create and add the variant objects in order:

```
Controller :: add(Variant* v) {
  variants.append(v);
}
```

Program 5.3 The `add` operation

A pointer to the current variant is maintained in the `Controller` and it points to the first member of the list initially, as shown in the code of the constructor. The variants in the list will be executed in order until one variant passes the acceptance test. The `Controller` implements the control in the following form:

```
int Controller :: action( ) {
  while(accepTest(currVariant.action( )) == ERROR) {
  undo( );
  currVariant = variants.next( );
  ... ... ... ...          //execute next variant
  }
}
```

Program 5.4 Implementation of the actual control

Finally, the `undo( )` operation can be implemented as a virtual function which is bound to certain low-level checkpointing services.

Degraded functionality. In practice, it is possible to implement different variants that provide the similar functionality with different levels of complexity or efficiency [Anderson & Lee 1981]. For example, in an adaptive recovery scheme the alternate actions for the primary can be some older versions of the primary and thus do not contain the faults that can be introduced by functional upgrades to the primary. Or these alternates may be designed deliberately to be less efficient in order to pass the acceptance test more easily. The primary usually attempts to deal with all possible input cases, whereas the alternates may handle only some of the input cases.

State restoration. The implementation of a particular fault tolerance scheme requires certain low-level support. For example, the state saving and restoration are essential for those schemes based on backward error recovery. This pattern implements state restoration operations as low-level services, which are transparent to the FTC programmers when they develop their software variants. The implementation of the `undo( )` can use the Memento pattern for checkpointing in [Gamma et al 1995] to save and restore the original state of the objects affected by the execution of the current variant. The operation may have to restore the state of the system, e.g. values of variables for the RB scheme or restore the state of all the external objects for a given CA action.

Adjudication functions. Two basic adjudication functions are the acceptance test for RB and voting for NVP. They can be used singly or in combination. Consensus recovery Block (CRB) and Acceptance Voting (AV) are two typical examples of the combined use. CRB votes the results first, and if it fails to find the majority, then an acceptance test will be used to find the correct answer. On the contrary, AV adjudicates the results first using an acceptance test, and only the results that pass the test will be voted to find the final answer. Similar to the reliable hybrid pattern [Daniels et al 1997], we can use the Composite pattern introduced in [Gamma et al 1995] to recursively combine various adjudication functions. Such combined instantiations permit a wide range of adjudication strategies, from the very simplest to highly complex solutions. Figure 5.12 shows the structure that allows recursive combination of the adjudication components. Take AV as an example again. The FT controller in our pattern needs to create an instance of the `Combined` component which, in turn, creates an `accepTest` object to test the results and creates a `simple voter` object to check the results that pass the acceptance test.

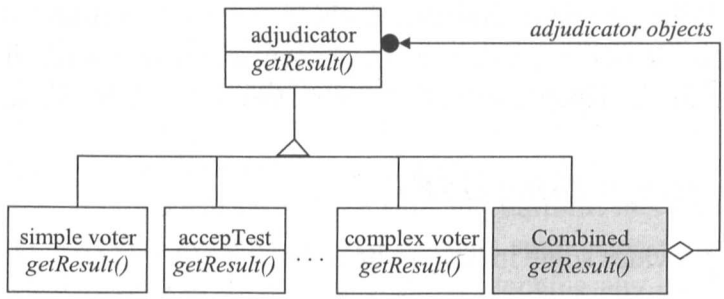


Figure 5.12 Recursive combination of adjudication functions

Reflective implementation. Following the reference architecture introduced in Section 5.1.3, it is quite straightforward to implement this pattern based on inheritance and delegation. However, our experience shows that a reflective implementation is also feasible when using a similar multi-level architecture. The meta level and base level can be treated as two separate levels in our reference architecture, each of which provides its own interface. For example, the base-level specifies the user interface for exploiting application functionality, and the meta-level defines the interface and components that determine the fault-tolerant behaviour of the application.

In a normal multi-level architecture, every level usually builds upon the levels below. There are no obvious bi-directional dependencies between two levels. In contrast with this, there are some mutual dependencies between levels in a reflective implementation. The base level builds on the meta level and vice-versa. For example, metaobjects can implement exceptional behaviour in case of an exception. However, this kind of exception handling must react according to the current state of computation. The meta level needs to retrieve the information from the base level, often from different components to those providing the interrupted service.

A reflective implementation can provide a greater degree of transparency than an implementation based on inheritance and delegation. It supports the change of fault tolerance schemes, statically or dynamically, even without modifying any source code of the fault-tolerant objects using the schemes. However, the reflective implementation is less efficient because of the complex relationship between the base level and the meta-level. Reflective capabilities often require extra processing including information retrieval, changing metaobjects, consistency checking, and communication between the two levels. All of these impose some performance penalty.

Known Uses

There has been a large amount of research and practical engineering in the area of fault-tolerant software, although in most cases they are critical applications such as aerospace systems, nuclear power plants and railway control systems (see the related survey in Chapter Two). The pattern itself arises partially from those known uses and partially from our practical experience over the years in developing fault-tolerant components for sequential and/or concurrent object-oriented software. It captures the most useful aspects of our experimental implementations that have in fact used a variety of programming languages including C++, Open C++, Ada 95 and Java.

5.2.4 Consequences of Using GSFT

The GSFT pattern provides some important **benefits**:

Desirable system-level support. All the desirable aspects expected from system-level support, as described at the beginning of this chapter, have been treated by the pattern thoroughly and quite satisfactorily. In particular, repetitive development effort is reduced greatly by introducing a large number of reusable interface components, reusable control mechanisms and low-level services. The development process is eased through a multi-level transparency mechanism. For example, each special type of programmer, the FTC users, the FTC programmers, or the FCC programmers, only have to address a limited set of concerns, ignoring any other implementation details. The application developer can specify, change or customise a special scheme for software fault tolerance through a simple interface without modifying any functionality-related source code. Different programmers may also choose the different number of variants and a special adjudicator depending upon application-specific requirements.

Improved dependability. The dependability of various approaches to fault-tolerant software has been studied extensively (see the related sections in Chapter Two). Academia and industry appear to have reached a general consensus that fault tolerance techniques for coping with software faults, if used properly, have the capability of increasing the entire dependability of a computer-based system [McAllister & Vouk 1996].

Transparency. The clients requesting fault-tolerant services are not aware of how fault tolerance is actually achieved. The fault-tolerant objects providing dependable services simply use the reusable control mechanisms to execute software variants and the adjudication function without knowing any implementation details of the mechanisms. The reusable control mechanisms may request further low-level services such as state restoration and synchronization, but do not need to know how services are implemented.

Flexibility. This pattern offers high flexibility. The choice of schemes, number and order of the variants can be determined dynamically in the pattern. Moreover, apart from support for a variety of fault tolerance schemes, the clients can also decide to use fault-tolerant services only when the need arises. This is because the clients have a reference to a fault-tolerant object but not directly to the generic FT interface. It is possible for different instances of the client to request different services including non-fault-tolerant one, i.e. only a single variant is executed without any adjudication.

Changeability and extensibility of components. If fault-tolerant objects change but their interfaces remain the same, there will be no functional impact on clients. Similarly, modifying the internal implementation of a fault tolerance scheme, but not the interface the generic FT interface component provides, will have no effect on both clients and fault-tolerant objects other than possible performance and dependability changes. The generic FT interface component provides a safe and uniform mechanism for changing and extending the software system. The components supporting a new fault tolerance scheme can be easily added to the system without touching any existing fault-tolerant objects. To use a new scheme, the fault-tolerant objects may have to pass a new parameter to the interface, but will not need to change any functionality-related source code.

Portability. This pattern hides operating system and network system details from clients, fault-tolerant objects and generic FT interface components by using a multi-level indirection architecture. When porting is needed, it will be sufficient in most cases to port the reusable interface components to a new platform and to recompile clients and fault-tolerant objects. If the low-level services hide system-specific details from the components located at higher levels of the reference architecture, only these low-level components need to be ported.

Reusability. First, the architectural pattern can be reused whenever software fault tolerance is required. Secondly, control mechanisms for a wide variety of fault tolerance schemes can be also reused. Thirdly, when building new client applications, even the functionality of the new application can be based on existing services. For example, if certain common services have been implemented as fault-tolerant objects, clients will not need to re-implement these services themselves. It may instead be sufficient to integrate these services into the new client application.

The generic software fault tolerance pattern imposes some **liabilities**:

Development and execution overheads. The implementation of different software variants and the adjudicator does lead to a high development cost. Thus, the use of this pattern can be justified only in systems that require a very high level of dependability. There is also a run-time overhead in terms of the execution time to perform the software variants, sequentially or in parallel, to switch over the variants, and to adjudicate the results.

Increased number of components. As there are different variants for a given function and the adjudication function, the size of the software increases. If an application system has constraints on the size of the software, introducing redundancy into the code may not be acceptable. This pattern also includes many more components than the components normally required by a non-fault-tolerant system. This may have a negative impact on system efficiency.

Restricted efficiency. Applications using this software fault tolerance pattern are usually slower than applications that do not use fault-tolerant services. Systems that depend directly upon a concrete fault-tolerant mechanism also give better performance than our general architecture, because the architecture introduces multiple indirection levels so that different concerns can be addressed separately and portability, flexibility and changeability can be improved. The performance penalty is also caused by the interaction between an increased number of components. A reflective implementation may require extra processing and coordination between the base level and meta level, which can decrease further the overall performance of the system.

Testing and debugging. Tested reusable components may ease the task of testing a client application developed based on these reusable components. However, debugging and testing a system involving many components can be difficult. For example, the cooperation between a client and a fault-tolerant object, or between a fault-tolerant object and a generic FT interface, can fail for various reasons – either the fault-tolerant object fails to deliver the requested service or something goes wrong on the communication path between the client and the fault-tolerant object.

Language support. Our reference architecture and pattern may be implemented using different implementation strategies and different programming languages. An implementation based on inheritance and delegation can use most object-oriented languages such as C++, Ada 95 and Java. However, a reflective architecture may be hard to implement in some languages, such as C++, which offers little or no support for reflection at all. It is almost impossible in such languages to exploit the full power of reflection, for example, adding new methods to a class dynamically. Limited capabilities of reflection could be added on an existing language. Open C++ is one of the examples.

Related Patterns

The NVP scheme has been previously implemented in various different ways, and one particular implementation is documented as Master-Slave pattern in [Buschmann

1995]. Another special implementation of RB is identified as Backup design pattern in [Subramanian & Tsai 1995]. Largely based on the object-oriented approach to software fault tolerance discussed in our previous work [Xu et al 1995c], [Daniels et al 1997] describes a pattern, called the Reliable Hybrid Pattern, to support the development of fault-tolerant applications that use NVP, RB or other advanced hybrid techniques. [Ferreira & Rubira 1998] introduces a Metapattern called Software Redundancy to achieve fault tolerance based on a reflective meta-level architecture. Our pattern is much more general and not limited to a single possible implementation such as the reflective solution. It also documents existing solutions and experience in implementing fault-tolerant concurrent software.

5.3 Case Study: The Fault-Tolerant Production Cell

Our reference architecture in Section 5.1 and the associated architectural pattern in Section 5.2 determine the basic structure of the solution to the problem of building fault-tolerant software, but they do not specify a fully-detailed solution. They provide a scheme for a generic solution to a family of problems, rather than a prefabricated module that can be used "as it is". This scheme must be implemented according to the specific needs of a particular application in hand. In this section, we will use a realistic case study to investigate various design and implementation details specific to a concrete implementation. In particular, we will examine the feasibility of using CA actions as a structuring tool to design industrial safety-critical applications.

An industrial production cell model, based on a metal-processing plant in Karlsruhe, Germany, was first created by the FZI (Forschungszentrum Informatik) in 1993 [Lewerentz & Lindner 1995], within the German Korso Project, in order to evaluate and compare different formal methods and to explore their practicability for industrial applications. Since then, this original case study, Production Cell I, has attracted wide attention and has been investigated by over 35 different research groups and universities. In 1996, the FZI presented the specification of an extended model of the original production cell, called the "Fault-Tolerant Production Cell" or Production Cell II [Lötzbeyer 1996]. This second model, which has an additional press, extra sensors and warning light systems to facilitate component failure detection and fault tolerance, is much more complex and realistic than Production Cell I. Unlike the first model, failures of electro-mechanical components and sensors in Production Cell II are of major concern. The cell is required to provide continuous service even if one of the two presses is out of order.

The original, rather simplistic, production cell model assumes no device or sensor failures occur. Under such assumptions, we used the CA action concept to organize and design a control program, and implemented it in Java [Zorzo et al 1999]. The control program that we had developed was then applied to a FZI-provided Tcl/Tk simulator, demonstrating how functional and safety-related requirements could be separately

satisfied by controlled multi-threaded cooperation and the strict enclosure of interaction between cooperating devices.

The Fault-Tolerant Production Cell exposes more and richer issues related to failures and fault tolerance, and it is therefore a valuable case study for investigating and developing concurrent fault-tolerant software. Because devices, sensors, actuators and the control program itself can fail, the required control program is necessarily much more complex, hence more realistic, than the program that we developed for the original, non-fault-tolerant production cell. Many dependability-related issues must be addressed properly.

Following a brief description of the Fault-Tolerant Production Cell model, we provide an analysis of software faults and possible hardware component failures. Based on the failure analysis, we will describe a design for a control program that uses CA actions to deal with both safety-related and fault tolerance concerns, and outline an implementation of the control program. Finally, we will discuss our experience with, and lessons learnt from, this practical case study.

5.3.1 Description of the Fault-Tolerant Production Cell

The Fault-Tolerant Production Cell consists of six devices: two conveyor belts (a feed belt and a deposit belt), an elevating rotary table, two presses and a rotary robot that has two orthogonal extendible arms equipped with electromagnets (see Figure 5.13). These devices are associated with a set of sensors that provide useful information to a controller and a set of actuators via which the controller can exercise control over the whole system. The task of the cell is to get a metal blank from its "environment" via the feed belt, transform it into a forged plate by using a press, and then return it to the environment via the deposit belt.

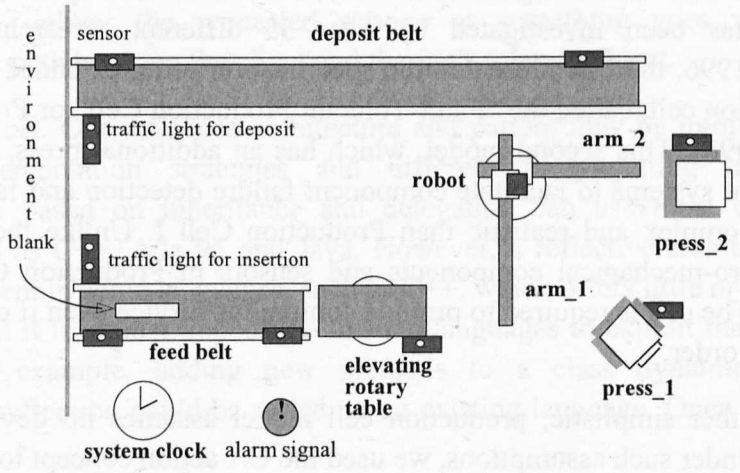


Figure 5.13 The Fault-Tolerant Production Cell (top view)

More precisely, the production cycle for each blank is: i) if the traffic light for insertion shows green, a blank may be added, e.g. by the blank supplier, to the feed belt from the

environment, ii) the feed belt conveys the blank to the table, iii) the table rotates and rises to the position where the magnets of the robot are able to grip the blank, iv) arm 1 of the robot picks the blank up and places it into an unoccupied press, either press 1 or press 2, v) the chosen press forges the blank, vi) arm 2 of the robot removes the forged plate from the press and places it on the deposit belt, and vii) if the traffic light for deposit is green, the plate may be forwarded further and carried to the environment where a container may be used, e.g. by the blank consumer, to store the forged pieces. (Normally both presses are used and a certain amount of interleaving of two such production cycles, one for each press, is possible.)

The controller that controls the entire cell can be implemented in hardware, and/or software. In this thesis we will investigate a software-implemented controller only. Our design and control program will support a varying, adaptive operating sequence of the robot in order to achieve high flexibility.

Normally both presses are used. Assuming that the blanks arrive at the feed belt frequently enough, the Production Cell II model specifies two typical operating sequences of the robot. The movement of the robot for interacting with press 2 can be exactly the same as that defined in Production Cell I [Lewerentz & Lindner 1995]. That is, i) arm 1 picks up a blank from the table, ii) arm 2 picks up a forged blank from press 2, iii) arm 2 places the forged blank on the deposit belt, and iv) arm 1 drops the unforged blank into press 2. However, since press 1 is closer to the table, the appropriate operating sequence of the robot for interacting with press 1 is slightly different: i) arm 2 picks up a forged blank from press 1, ii) arm 1 picks up an unforged blank from the table, iii) arm 1 drops the blank into press 1, and iv) arm 2 places the forged blank on the deposit belt.

The above two sequences of the robot are normally executed alternately. If one of the presses is out of order, the corresponding operating sequence is skipped. In more complicated cases, e.g. uncertain arrival of blanks from the feed belt, the movement of the robot is required to be more intelligent and adaptive. A variety of operating sequences will be possible, depending upon how the control program is designed.

Basic System Requirements

A control program and the controlled cell must satisfy certain requirements specified by the Fault-Tolerant Production Cell model, namely:

Safety. i) device mobility must be restricted, ii) device collisions must be prevented, iii) blanks must not be dropped outside safe areas (i.e. feed belt, table, press, and deposit belt,) and iv) sufficient distance must be maintained between blanks.

Liveness. Any blank put into the cell via the feed belt must eventually leave the cell via the deposit belt and have been forged by one of the presses. In addition, this property must still hold if only one of the two presses fails.

Failure Detection and Continuous Service. When any of a large number of defined failures occurs, it must be detected and unless it just concerns one of the presses the system must be stopped in a safe state. After recovery from the failure, which typically would require action by the user of the Production Cell, the system should be able to resume operations starting from this safe state. Similarly, after a failed press has been repaired, it should be able to resume its contributions to the production process. (Certain safety requirements can no longer be met if some special failures occur, e.g. a blank is dropped outside safe areas, but other safety properties must still be guaranteed, e.g. restricted device mobility.)

Other requirements such as flexibility and efficiency may be taken into account, but must not conflict with the above requirements.

System Clock and Stopwatches

The Fault-Tolerant Production Cell model provides a global system clock that gives the current time at any instant. Based on the system clock, a control program can implement several stopwatches supervising certain processes, e.g. the movement of the feed belt. Whenever the system starts or re-starts, the system clock is initialized. The current time is transmitted with every transmission of sensor values. In order to help detect certain failures, upper time bounds on the movement of some devices are defined and can be checked using stopwatches. Once the timeout expires, a failure will be reported. (For example, $t_{feedbelt}$ is defined as the maximum time during which the feed belt carries a blank from the start to the end of the belt, and a typical value for $t_{feedbelt}$ is 50 seconds [Lötzbeyer 1996].)

Alarm Signals

The Fault-Tolerant Production Cell model also provides an alarm signal mechanism for reporting component failures to the user of the Production Cell. The control program is required to switch on the alarm signal whenever a hardware failure is detected. (In our design, the alarm can be also switched on in order to indicate the occurrence of a software fault in the control program itself.) An attached message can be sent to the user with more detailed information about hardware failures and software errors. The model assumes that this alarm signal can be turned off only by the operator, indicating that all faulty devices, sensors, actuators or software faults have been repaired.

5.3.2 Assumptions and Failure Analysis

Before analyzing possible software faults and component failures of the cell, we state the major assumptions made in the Fault-Tolerant Production Cell model, as defined by FZI:

Assumption 1: The system clock, two traffic lights, and the alarm signal mechanism are fault-free and do not fail.

Assumption 2: Values of sensors, actuators and clocks are always transmitted correctly without any loss or error.

Assumption 3: No failure can cause devices to exceed certain limiting positions; in the worst case devices are stopped automatically.

Assumption 4: All sensor failures are indicated by sensor values. Boolean sensors return a zero value, and enumeration type sensors return a specified value that indicates a failure.

Assumption 5: All actuator failures will cause devices to stop.

Software Faults in the Control Program

A control program for the Fault-Tolerant Production Cell is a complex program with multiple concurrent control threads that must coordinate the interaction of multiple concurrent devices in a highly safe manner. Due to the complexity, it cannot be simply assumed that the control software itself will be free of error. Software faults might manifest themselves during the system operation time. Especially for addressing safety-related concerns, software faults must be taken into account and be handled properly by means of various techniques discussed in previous chapters.

Apart from possible incomplete or inconsistent specifications of computation requirements, there are two other sources which may introduce software bugs into the control program. One source is the selection of inadequate or insufficient algorithms which do not cover all realistically possible application situations, especially those rare but possible system conditions. Another source is mistakes in converting a selected algorithm into a program for a specific environment such as the Fault-Tolerant Production Cell. In addition, since the control program involves concurrency and/or distribution, software faults can be further introduced into the part that integrates and coordinates multiple concurrent modules. Although individual modules may meet their respective specifications, the specifications are often incomplete and inaccurate as to coordination and cooperation between modules.

Software fault avoidance is particularly difficult in realistic safety-critical applications. Although the use of formal methods is very helpful in improving the software quality, formal verification of designs and implemented programs has not yet advanced to the level where the absence of any error in sizeable software can be verified by machine. (This topic will be addressed further in Section 5.3.4).

The Production Cell can be viewed as the environment of a control program, and component failures of the cell can be regarded as environmental failures with respect to the control program. For a given device of the Production Cell, we classify possible component failures into: i) sensor failures, ii) actuator failures, and iii) lost or stuck blanks. We also show how a given failure can be detected by sensors, actuators, stopwatches, singly or in combination. It is important to notice that in many cases

different types of failures cannot be distinguished just based on the on-line information available. We therefore discuss failure detection only, and assume that fault diagnosis and subsequent device repair are performed off-line. In the interests of simplicity and brevity, our discussion just treats the case of failures of either the robot or a press; for a complete treatment, see [Xu et al 1998b].

Failures of the Robot

Sensor Failures. There are three sensors associated with the robot — each sensor returns one of several pre-defined values about the position of one of the robot's arms or the robot's rotary position. Three electric motors are responsible for rotating the robot or extending/retracting its arms. Sensor failure or electric motor failure is indicated automatically by a special sensor value, but these two types of failures cannot be distinguished using this sensor value alone.

Actuator Failures. There are three kinds of actuator associated with the robot and each has its own failure modes: i) failure modes of actuators that retract an arm of the robot include: no response (i.e. cannot move) and unexpected stopping of a moving arm, which can be detected by checking values of robot sensors, ii) failure modes of actuators that switch an arm magnet on or off include: no response (e.g. the arm cannot pick up or cannot drop a blank) and unexpected picking or dropping, which can be detected only by checking values of other devices interacting with the robot, and iii) failure modes of the actuator that rotates the robot, that include: no response (i.e. cannot rotate) and unexpected stopping of the rotating robot, which can be detected immediately by checking values of the sensor that indicates the robot's rotary positions.

Lost Blank. This type of failure can be detected only by checking a group of sensor values from various devices interacting with the robot.

Failures of a Press

Sensor Failures. There are four sensors associated with this press, one reporting whether a blank is in the press (called *blank sensor*), and others reporting press positions. A failure of the blank sensor can be detected by checking whether a robot arm has transferred a blank to or from the press. The failure of a sensor that reports press positions can be detected by using a stopwatch to measure the moving time of the press and by checking other sensor values on press positions.

Actuator Failures. Failure modes of the actuators that move the lower part of press 1 include: no response (i.e. cannot move) and a moving press unexpected stopping, which can be detected by checking values of the press position sensors and values of stopwatches.

Stuck or Lost Blank. This failure can be detected only by checking the value of the sensor that reports whether a blank is in a press.

Error Detection Measures

In order to detect software errors, appropriate detection measures must be incorporated into a software design. For this Production Cell application, any control commands sent to the cell cannot be simply recalled. For example, a wrongly forged blank cannot be easily restored. In some cases, the possibility of using backward error recovery in an effective way is rather limited. It is therefore important to detect errors before the control program actually sends a command to the cell. In fact, the practical effectiveness of any scheme for coping with software faults rests critically on its ability to detect software errors before they can impact on the system environment. Assertion statements are a common form of error detection measure. Other detection mechanisms and measures may be used as well. By implementing different schemes such as adaptive recovery and fault masking, we can use acceptance tests and voting checks to mitigate the fallibility of run-time assertions. The use of nested actions can provide further protection — an error which is not detected at a low level might be identified by a detection mechanism or measure at a higher level.

These mechanisms and measures can also be used to detect hardware component failures of the cell. For example, after the control program has sent a control command to the robot and asked the robot to drop a blank into press 1, the value of the sensor that reports a blank in the press must be checked by an assertion statement. If the sensor returns 0, indicating that no blank in press 1, then an exception must be raised.

However, there are several possibilities that could have caused this exception: i) the blank might have been lost, ii) arm 1 of the robot might have failed to drop the blank, and iii) the sensor of press 1 might have failed to report that the blank has been dropped into the press. If a powerful on-line diagnosis algorithm could identify this failure as the sensor failure, exception handling and error recovery would be quite straightforward — just report the exception to the user and continue normal operations of the cell. However, our analysis shows that distinguishing these failures from each other at run-time is extremely difficult, if not impossible. In most cases, if a failure occurs and thus an exception is raised, the cell will simply have to be stopped in a safe state, if at all possible, for the user to deal with. (Certain safety requirements cannot be met if a blank is dropped outside the safe areas, but the others must still be maintained.)

Failures of sensors that report press positions and failures of the press actuator can be detected by assertion statements and identified unambiguously with the aid of stopwatches. Such failures must be reported to the user through the alarm. However, because the Fault-Tolerant Production Cell has two presses, normal operations can be maintained using a single press, albeit with some performance degradation.

A fault-tolerant program should have the ability to confine damage and failures. For the production cycle of the cell, a device or sensor failure should not affect normal operations of other devices. For example, when a failure of the robot occurs and is handled by the control program, the deposit belt should still deliver an already forged blank, if there is one, to the blank consumer. In the following, we will demonstrate how

CA actions can confine damage and failures effectively, and minimize the impact of component failures on the entire cell.

5.3.3 Design of a Control Program Using CA Actions

As discussed in Section 5.3.1, a control program for controlling the Fault-Tolerant Production Cell must satisfy a number of requirements regarding safety, liveness and correctness. Moreover, these requirements must be met even when one of the two presses has failed. The main characteristics of our CA action-based design are the way it separates safety, functionality, and efficiency concerns among a set of CA actions, which thus can be designed, and validated, independently of each other, and of the set of device/sensor-controllers that dynamically determine the order in which the CA actions are executed at run-time. In particular, the safety requirements are satisfied at the level of CA actions, while the other requirements are met by the device/sensor-controllers. There is a detailed discussion in [Zorzo et al 1999] as to how these design decision were made and why we used certain actions to enclose the interaction between certain devices in our control program for Production Cell I. Our design for Production Cell II follows a similar strategy. It includes 12 main CA actions; each action controls one step of the blank processing and typically involves passing a blank between two devices. Any device can move only within a CA action. (An action can contain further nested actions — see Figure 5.15 for an example.)

There are six concurrent execution threads in the control program, corresponding to the six devices: `FeedBelt`, `Table`, `Robot`, `Press1`, `Press2`, and `DepositBelt`, each of which threads basically performs a simple endless loop. All device movements are performed within CA actions, and the devices involved in each action are switched off before the action is left, so that when not under the control of an action each device is stationary. Two additional threads model activities in the environment: `BlankSupplier`, and `BlankConsumer`. Note that `FeedBelt` is responsible for controlling the traffic light that indicates when another blank can be inserted, while `BlankConsumer` is responsible for controlling the light that indicates when a processed blank can be deposited. A blank is designed as an external object with respect to the top-level CA actions. Usually, one role of a CA action takes the blank as an input argument, and the device corresponding to this role passes it to another role which returns it as an output argument.

Figure 5.14 portrays the 12 related CA actions as overlays on the FZI simulator diagram [Lötzbeyer 1996]. Note that an intersection between CA actions in Figure 5.14, e.g. between `TransportBlank` and `LoadDepositBelt`, represents the fact that those CA actions cannot be executed in parallel. The mutual exclusion feature of CA actions guarantees that a blank or a device cannot be involved in more than one action at a time so that neither blanks nor devices can collide. Furthermore, even if the actions that devices participate in are invoked in the wrong order, because of a control program design fault, then the result will be at worst a safe deadlock.

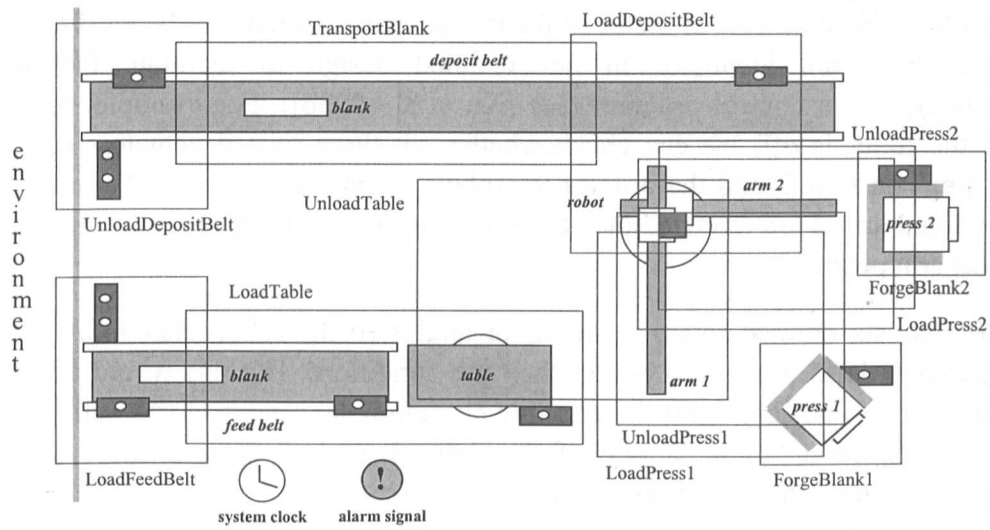


Figure 5.14 CA actions that control the Fault-Tolerant Production Cell

As mentioned previously, each hardware device is associated with a device-controller (i.e. an execution thread) which is responsible for dynamically specifying the sequence of actions that the device will participate in. For example, without compromising safety and functionality requirements, the `robot` thread can skip all the CA actions related to one of the presses if this press has failed, and so tolerate this fault.

Design of CA Actions

Our design assumes that an action will begin only if its pre-conditions are valid, and that if no exception is raised during the execution of an action then its post-conditions will hold (though this could, if so wished, be checked using an acceptance test). In the following, we first address the normal pre- and post-conditions for actions that control the entire cell. For a given action, these conditions are used to ensure that the execution of that action will not violate in any way the system requirements given in Section 5.3.1, especially those related to safety and fault tolerance. Due to limitations of space, we take just the action `LoadPress1` as an example.

CA action `LoadPress1`

Pre-conditions	*Post-conditions*
robot off	robot off
blank on arm 1	no blank on arm 1
both arms retracted	both arms retracted
robot at one of defined angles	robot angle: arm 1 towards press 1
press 1 off	press 1 off
no blank in press 1	blank in press 1
press 1 in bottom position	press 1 in middle position

Table 5.2 Pre- and post-conditions of CA action `LoadPress1`

Values of the related sensors or states of the related actuators that can be used to check these conditions are identified in our detailed design to facilitate the actual implementation of a control program (see [Xu et al 1998b]). For example, to check whether the robot is off, we can check whether all three related actuators (i.e. that retracts arm 1 or arm 2, and that rotates the robot) are in the stop state. To make sure that arm 1 and arm 2 are retracted (a safe state), we can check values from the sensors that report arm positions.

The robot has six defined rotary positions or angles so the robot-related CA actions could specify a defined angle as one of their pre-conditions. But this would affect the flexibility of the robot and limit the possible execution sequences of CA actions. The weaker pre-condition "robot at one of the defined angles" permits more possible execution sequences, thereby improving system performance.

We will now show how CA actions can deal with various types of failures in a well-controlled manner (e.g. by specifying the exceptional post-conditions for a given action, as shown in Section 5.3.5). Consider the action `LoadPress1` again. Figure 5.15 illustrates the interactions (themselves involving nested CA actions) between the participating threads within the `LoadPress1` action. This action has four roles: `Robot`, `Press1`, `RobotSensor`, and `Press1Sensor`,. and represents the cooperation that arranges for arm 1 of the robot to drop a blank into press 1.

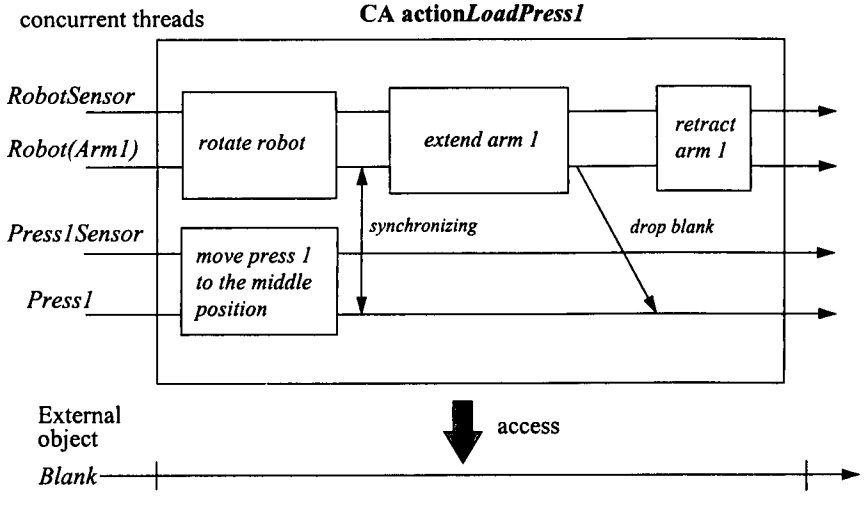


Figure 5.15 CA action `LoadPress1`

In the form of a specification language, action `LoadPress1` is described below using the COALA notation, which was developed for the formal specification of CA actions [Vachon et al 1998]. (Our Java implementation of the control program is based on a set of pre-defined components for CA actions that can be used to implement CA action designs specified in COALA.)

```
CAA LoadPress1;
Interface
    Use
      MetalBlank;                                    //specify external objects
    Roles
      Robot: blankType, robotActuator;
      Press1: blankType, press1Actuator;
      RobotSensor: arm1ExtensionSensor, robotAngleSensor;
      Press1Sensor: blankSensor, lowPositionSensor, midPositionSensor;
    Exceptions
      Press1Failure, Arm1Failure1, ...;              //exceptions to signal
End LoadPress1;
```

Program 5.5 Interface of CA action `LoadPress1`

The exceptions declared in the **`Interface`** part of an action are those that can be signalled to the enclosing action. The roles of an action can signal an exception directly but must guarantee that the exception that is signalled has been agreed by all the roles of that action. In the case of abortion or failure, the CA action support mechanism (which can be assumed by the application programmer to be fault-free) will enforce the abortion and signal the appropriate exception, either `abort` or `failure`, to the enclosing action.

```
CAA LoadPress1;
Body
    Use CAA                                          //specify nested actions
      RotateRobot, MovePress1toMid, ExtendArm1, RetractArm1;
    Object
      robotPress1Local: Local;                       //shared local objects
    Exceptions
      press1_failure, blank_sensor_failure, ...; //internal exceptions
    Handlers
      press1_handler, blank_sensor_handler, ...;
    Resolution
      press1_failure -> press1_handler, ...;  //resolution graph
    Role Robot(...);
    Role Press1(...);
    ...
End LoadPress1;
```

Program 5.6 Body of CA action `LoadPress1`

Exceptions declared within the **`Body`** of a CA action can be raised by roles. When multiple exceptions are raised within an action, the CA action support mechanism controls the execution of a resolution algorithm based on an exception resolution graph declared in the **`Resolution`** part. After a resolving exception is identified, the corresponding handler declared in the **`Handlers`** part will be invoked.

An exception handler will attempt to bring the system back to normal. If it is successful, the CA action will end with a normal outcome. However, in most situations

the handler can only provide some degraded service, i.e. an exceptional outcome, and must signal the corresponding exception. Again, in the case of abortion or failure, the CA action support mechanism will take control. If a further exception is raised during the execution of an exception handler, control is transferred to the CA action support mechanism immediately and the action must either abort or signal a `failure` exception.

Design of Device-Controllers

Given a set of CA actions to control the interaction of devices in the Production Cell, device/sensor-controllers are used to determine dynamically the order in which the CA actions are executed. Eight controllers are designed: `FeedBelt`, `Table`, `Robot`, `Press1`, `Press2`, `DepositBelt`, `Supplier`, and `Consumer`. Two queue objects are defined in order to improve the flexibility of operations of both the robot and the deposit belt: `robotQueue` and `depositBeltQueue`. The `Press1` controller is shown below as a simple example:

```
Press1Controller:
  loop forever {
        robotQueue.put(PRESS1_FREE)              //put message in robotQueue
        LoadPress1.Press(plate)                  //activate action LoadPress1
        ForgeBlank1.Press(plate)                 //activate action ForgeBlank1
        robotQueue.put(FORGED_PLATE_IN_PRESS1)   //put message in robotQueue
        UnloadPress1.Press(plate)                //activate action UnloadPress1
  }
```

Program 5.7 The `Press1` controller

Figure 5.18 shows the interactions between the controllers and CA actions, where boxes represent CA actions and ovals represent controllers. A grey line indicates message passing between controllers, while a black line connects an action to a controller or vice versa and implies that the controller plays a role in that action.

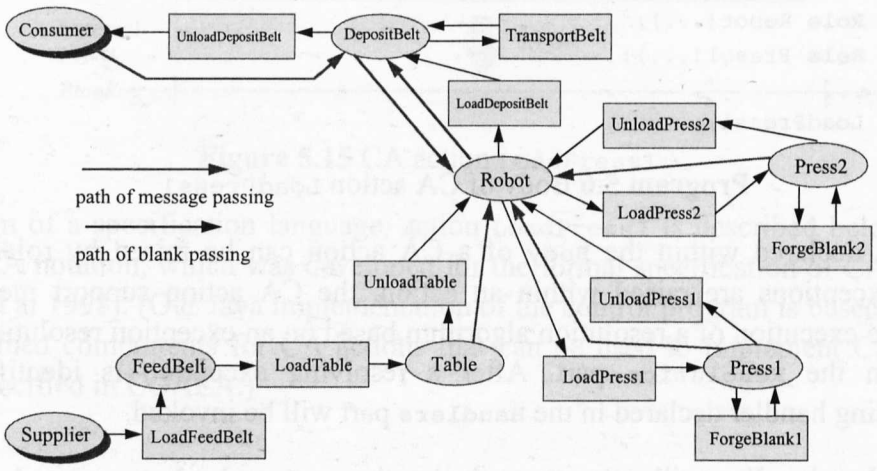


Figure 5.18 Interaction between controllers and CA actions

5.3.4 Dealing with Software Faults in the Control Program

We first investigate situations involving software faults only, i.e. we assume that *only faults that can occur are software faults in the control program, and all the hardware components of the Production Cell are fault-free.* (More complex situations involving hardware component failures will be discussed later.)

The major software faults that may remain in the control program include i) interaction faults that occur when the interaction relationship between roles of a CA action, between interacting CA actions, or between the control program and the Production Cell has not been specified properly or analyzed sufficiently, and ii) timing faults that occur when an operation, a role or a CA action is not completed in a pre-specified amount of time.

Software faults can be also involved in error detection and recovery mechanisms for CA actions. They may cause a mechanism to handle an error improperly or may invoke the mechanism when no error exists. We regard these mechanisms as part of a CA action support mechanism, and assume that the support mechanism itself has been tested extensively and is fault-free. To provide support for this assumption, these mechanisms are implemented as reusable and well-tested components in our GSFT pattern. The CA action support mechanism also contains a concurrency-control mechanism for controlling access to external objects. In the Fault-Tolerant Production Cell, the external objects (i.e. metal blanks) cannot be shared by two or more concurrent CA actions due to safety-related concerns. We choose to use a simple concurrency-control mechanism to minimize the probability of residual software bugs. This mechanism allows a monitor object to get the state information of a blank concurrently with a running CA action.

Software faults such as interaction faults and timing faults have not been addressed adequately in the initial requirements for the Fault-Tolerant Production Cell [Lötzbeyer 1996], while the early high-level COALA specification for implementing the control program (see Section 5.3.3) focused mainly on hardware device and sensor failures. For these reasons, and the inadequacy of available fault removal techniques, we had to admit that despite our best efforts to the contrary, software faults might exist in our control program. This indeed turned out to be the case since we have observed subsequently that some software design faults occur while our control program is in use. For example, a transient software fault occurs in the FZI simulator when arm 1 of the robot is required to place the blank into an unoccupied press. The arm performs most specified operations but it fails to drop the blank into the press. This fault manifests itself only occasionally and makes its removal extremely difficult. Another software bug appears in our control program in the form of interaction fault. This interaction fault manifests itself only when more than two blanks are placed into the system. Under certain conditions at run-time, two interacting CA actions can be involved in a deadlock situation from which no further operations are possible.

It is therefore evident that we have to deal with remaining software faults that may occur while the Production Cell is in operation. There are a variety of software fault tolerance schemes we may take into account. For transient software faults, a simple "re-try" strategy is often effective [Gray 1990]. For example, the `LoadPress1` action (see Figure 5.15) may have dropped a blank into press 1 properly, but the nested action `RetractArm1` failed to retract the arm to a correct position. Instead of attempting to execute an action variant, the recovery operation can simply re-execute the nested action in the hope that the same error may not recur. If the control program is executed in a distributed environment, the nested action in question may be re-executed on a different node of the distributed system. Timing faults can be treated by a simple timeout mechanism associated with the CA action support mechanism. For a given task (e.g. the execution of a role or an action) we specify a pre-defined amount of time. If the task is not completed in the time, a timeout exception will be raised. Since there is no strict timing constraints in the initial requirements for the Fault-Tolerant Production Cell, the recovery measure can be quite simple, such as re-execution and abortion.

However, for most software design faults (e.g. the interaction fault in our control program) a rollback and re-try approach is insufficient. Instead, tolerance of such software faults must rely on the application of design diversity. We want to use design diversity to minimize the probability that the independently designed variants contain similar errors that can cause the variants to fail simultaneously. (Although it is possible introduce diversity into the specification and other phases of the system life cycle, in this case study we will focus on the use of diversity in the design phase and the implementation phase.) While random diversity may be achieved by different programmers and designers, we feel that diverse data structures and algorithms are less likely to fail simultaneously. Such an approach is often called "enforced diversity" which enforces systematically the use of diverse structures and algorithms in different program variants. For example, based on our CA action-based design of the control program, two software variants for the `LoadPress1` CA action are developed using enforced diversity. Variant One is implemented in the same form as that shown in Figure 5.15, which involves several concurrent activities and two concurrent nested actions `RotateRobot` and `MovePress1toMiddle`. Variant Two is designed to provide the identical functionality but following a simpler algorithm without any concurrency. Within the second variant, five nested CA actions are executed sequentially in the order of `MovePress1toMiddle`, `RotateRobot`, `ExtendArm1`, `DropBlank`, and `RetractArm1`. Because the interaction relationship between these nested CA actions is essentially diverse in two variants, the probability that the variants fail identically should be reasonably low.

We discussed two particular techniques for incorporating design diversity into CA actions in Section 4.4.2: Adaptive Recovery (AR) and Fault Masking (FM). The AR approach uses a container CA action as a form of controller. The container action uses an acceptance test and other assertion statements to detect software faults at run-time and performs the test on the state of the related external objects that may have been changed by the first (nested) action variant. Once an error is detected, the container

action will roll the external objects back to the previous, correct state and invoke the second action variant.

The FM approach also uses a container action to control the execution of *n* (nested) action variants. These variants are executed in parallel, performing required operations on the related external objects. For each of these external objects, *n* clones must be generated for *n* action variants. Each participating thread of the container action must be forked into *n* sub-threads which will participate further in *n* variants respectively. When these variants are complete, the container action must decide the correct state of external objects based on the state of all the clones. Before the container ends, the sub-threads are merged and redundant clones discarded. (Notice that both threads and external objects can be replicated at the system level, i.e. outside the container action. This may simplify the responsibility of a container action, but the related overheads are added to the level of the overall system.)

Now we have to decide which of the above schemes should be used in 12 main CA actions of the control program. We have found that the AR approach is more appropriate due to the following considerations:

1) The AR scheme introduces relatively low additional complexity without having to make clones of the external objects and to replicate threads. In our design, each action performs certain operations of the blank object and other device objects. The device objects such as `Robot` and `Press1` are treated as a special form of the external objects, i.e. the hardware objects in the environment of our control program. The atomicity of these device objects is still maintained because all device movements must be performed within CA actions and a device must be stationary whenever it is not under the control of an action. However, making clones of these device objects are extremely difficult, if not impossible, and would be very expensive if additional hardware devices are required.

2) The full strength of the FM scheme requires the use of multiple processors and the mechanism for forking and merging threads. It is particularly suitable for a system with stringent real-time requirements. The AR scheme provides more flexibility since it allows the control program to run on a single processor, but it may be at a disadvantage in a real-time system. However, the Fault-Tolerant Production Cell has no any strict timing constraints. Most of the time, the AR scheme executes only the first action variant. The occasional execution of a backup variant is quite acceptable for this particular application. For normal external objects the state saving and restoration required by the AR scheme impose only a small run-time overhead (see Section 4.4.3). The device movements for the purpose of error recovery may take a longer time than the state restoration of a blank object, but they do not compromise the safety concerns since all recovery operations must be performed within the related CA action.

3) Unlike the FM approach, the AR scheme uses an acceptance test to validate a single result at a time without having to examine all the results produced by multiple variants. However, we have to keep the test reasonably simple and reduce the probability that the test itself contains software faults. In our design, we have developed sets of pre- and post-conditions for each main CA action. All the required acceptance tests for CA actions are derived from the corresponding post-conditions. The correctness of these tests were also validated when our design was examined by extensive formal verification and model-checking. The error-detection coverage is further improved by a large number of executable assertion statements within CA actions and run-time checks supported by the hardware platform. Some assertion statements may be switched off when the system performance has to be increased.

Figure 5.16 illustrates the control structure of the `LoadPress1` CA action using the AR scheme to tolerate software faults. CA action `LoadPress1` is designed as a container action which contains two diversely designed variants. Normally, the container action just executes the first variant. If no error is detected, then the container action ends with a normal outcome. However, if an error is detected by either an assertion statement or the acceptance test (at the end of the first variant), this error must be reported to the container action.

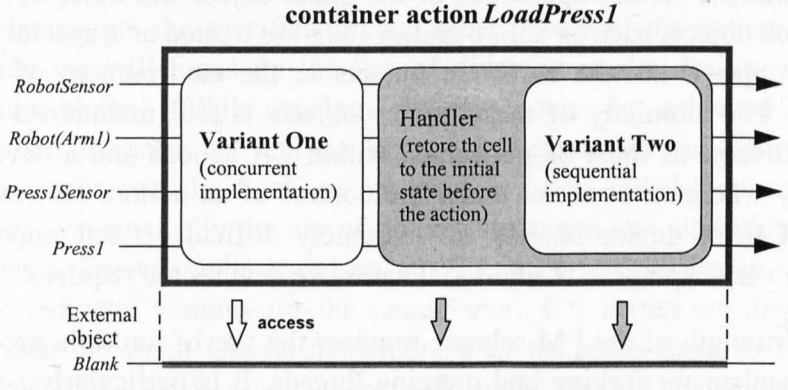


Figure 5.16 `LoadPress1` as container action to tolerate software faults

In principle, error propagation and error recovery should be performed within a structuring framework for exception handling. When an error occurs in Variant One, an appropriate exception must be signalled from the variant to the container. The corresponding handler will then be called, which has to restore the related device objects and the blank object to their original state — the state before the execution of action `LoadPress1`. After finishing the restoration, the handler will invoke the second variant in the hope that the same software error will not occur again. In the worst case that an error takes place again, it is always possible for action `LoadPress1` to signal an `abort` exception, if the sate restoration is successful, or a `failure` exception to its containing action. By way of example, we consider the arm 1 position error again. A handler must handle the exceptional situation where the nested action `RetractArm1`

fails to place the arm in a correct position. This handler can use both a re-try strategy and a diversely designed variant for the action. We outline the basic requirements for the handler as follows.

> *Handler for Arm 1 Position Error:* Re-execute the `RetractArm1` action and check the arm position. If the same error persists even after a pre-determined number of re-executions, then restore both the device objects (i.e. `Robot` and `Press1`) and the blank object to their original state. Invoke the second variant. If the variant signals an exception, instead of a normal outcome, then restore the state again. If the state restoration is successful, signal an `abort` exception to the container action, otherwise signal a `failure` exception.

Our design of the control program uses 12 main CA actions to address the safety requirements. The application of software fault tolerance techniques at this level enhances the ability of the control program to handle software errors so as to improve both system reliability and safety. Our design also establishes a set of device/sensor-controllers on the top of these CA actions to coordinate the execution order of the main actions. Conceptually, this device/sensor-controller level may be considered as a special, systemwide CA action. This outermost CA action starts when the control system begins, and it ends when the system stops normally according to user requirements. To achieve software fault tolerance at the level of the overall control system, all the schemes for sequential programs discussed in Chapter Two including recovery blocks and *N*-version programming can be applied directly by treating the outmost action as a single sequential system that encloses complex concurrent activities inside itself. Since we are concerned mainly with safety aspects in this case study, which have been addressed at the level of 12 main CA actions, we will not discuss further details of obtaining software fault tolerance at the level of the outermost system.

5.3.5 Dealing with Hardware Component Failures in the Cell

We now investigate situations involving single hardware faults, i.e. we assume that *only one component failure of the Production Cell can occur before the system stops, and the component is repaired.* Based on our previous failure analysis in Section 5.3.2, the following table shows the related failure modes of robot and press1 within the `LoadPress1` action and the means of detecting these failures. (Because the movement of arm 2 and the top position of press 1 are not involved in this particular action, arm 2-related failures and the failure of the sensor that reports the top position have not been listed in the table.) During the execution of a CA action, if any failure in Table 5.3 occurs and is detected by an assertion statement or an acceptance test, a corresponding exception will be raised within the action by one of its roles. The exception is propagated immediately to other roles of the action and all roles then transfer control to their exception handlers for this exception so that they can attempt to perform appropriate error recovery. In most cases when a component failure takes place in the

cell, it is not possible to recover completely from the error and the *normal* post-conditions of the action can no longer be satisfied. Thus, exceptional post-conditions with respect to various given failures must be defined to specify the exceptional outcomes of an action.

Failure modes of robot	*Failure detection*
Sensor failure	
position and rotary sensors	related values in these sensors
Actuator failure	
retract motor fails	retract sensor
arm 1 magnet fails	press 1 blank sensor
rotary motor fails	rotary sensor
Blank	
stuck or lost	press 1 blank sensor

Failure modes of Press1	*Failure detection*
Sensor failure	
blank and position sensors	other sensors and stopwatch
Actuator failure	
motor fails	position sensors and stopwatch
Blank	
stuck or lost	press 1 blank sensor

Table 5.3 Failure types of robot and press 1 and failure detection

By way of example again, we outline the basic requirements for the handlers of two typical exceptions:

> *Handler for the Press 1 Failure:* The `LoadPress1` action performs forward error recovery by moving the robot to an appropriate position so that it will be able to put the unforged blank, which is still on arm 1, into press 2 once the press is available.
>
> *Handler for the Rotary Sensor or Motor Failure:* (In this case, action `LoadPress1` fails to rotate the robot to the intended position.) The action will simply use backward error recovery to attempt to move the robot back to its initial position and rotate it again. If the failure persists, the action will produce an exceptional outcome as defined below.

For the `LoadPress1` action we identify seven exceptional outcomes and corresponding exceptional post-conditions (see [Xu et al 1998b]). By way of example, Table 5.4 illustrates two exceptional outcomes, i.e. those when press 1 or the blank sensor (that reports a blank in the press) failed. It is important to notice from the table that different exceptional outcomes may lead to different states of the cell. For example, the exceptional outcome caused by the press 1 failure corresponds to the situation where

the Production Cell continues with only one operational press. (In fact, without compromising safety and functionality requirements, the `robot` thread or robot-controller can skip all the CA actions related to press 1, and so tolerate this failure.) On the other hand, since the blank sensor is a redundant component of the cell, if both presses are still operational its failure merely requires a report to be made to the user of the cell. However, the other five exceptional outcomes will have to stop the entire cell in a safe state.

Exception to signal	*Exceptional post-conditions*
Press1 failure	robot off
	blank on arm 1
	both arms retracted
	robot angle: arm 1 towards press 2
	press 1 off
	no blank in press 1
Blank-sensor failure	robot off
	no blank on arm 1
	both arms retracted
	robot angle: arm 1 towards press 1
	press 1 off
	blank in press 1
	Press 1 in middle position

Table 5.4 Two examples of exceptional post-conditions

By means of such analyses, given the way in which CA actions enable the different failure situations to be treated independently of each other, the design of the actual set of handlers for the various exceptional outcomes of each of the 12 top-level CA actions becomes rather straightforward — some details can be found in [Xu et al 1998b].

Dealing with Concurrent Hardware Failures

Now let us address the problem of possible concurrent failures. In the interests of simplicity, we assume that *only two failures may occur within the same time interval before the system is stopped and the related components repaired.*

In the Fault-Tolerant Production Cell, some concurrent sensor and device failures can be covered implicitly by the corresponding single failure situation. In these cases, the error recovery measure for the single failure situation will be sufficient for handling concurrent hardware failures. However, most of concurrent sensor and device failures must be handled respectively and separate post-conditions must be specified. The following table shows exceptional post-conditions for an example of concurrent sensor and device failures:

Exception to signal	*Exceptional post-conditions*
(Rotary sensor or motor failure) & Press1 failure	robot off
	blank on arm 1
	both arms retracted
	press 1 off
	no blank in press 1

Table 5.5 Exceptional post-conditions as to concurrent failures

The failure of the robot's rotary sensor or motor can be detected automatically and indicated by a special sensor value. However, the returned sensor value does not indicate which component, i.e. the sensor or the motor, actually failed. This causes difficulty in performing effective error recovery. Very often, despite a failure having been detected, it is not possible to determine from the available sensor readings which of several possible failures has actually occurred. In such circumstances the control program is designed simply to bring the system to a stop in a safe state, so that off-line diagnosis can be performed. However, where feasible, online diagnostic programs can be used to identify actual failures, e.g. decide whether it is a press sensor failure or a press motor failure by a combined use of sensors, the actuator and stopwatches.

Since the Fault-Tolerant Production Cell model assumes that sensor readings are always correct and accurate, errors caused by software design faults can be easily distinguished from hardware component failures which are indicated immediately by certain sensor values. If an error is detected but no sensor value indicates a hardware component failure, our design will treat the error as the manifestation of a software fault, and will use appropriate software fault tolerance measures to perform error recovery.

For each (enclosing or nested) action, various exceptions are defined based on failure analysis and an exception graph for resolving concurrent exceptions is defined. For example, the `LoadPress1` action may give rise to exceptions such as `pr1_failure` (press 1 failure), `b_sensor_failure` (blank sensor failure), `arm1_failure1` (blank lost), `arm1_failure2` (cannot drop the blank), `rs_m_failure` (rotary sensor or motor failure), `as_m_failure1` (arm 1 sensor or motor failure while the blank on arm 1), `as_m_failure2` (arm 1 sensor or motor failure while the blank in press 1), `cs_failure` (control software failure(s)), and `rt_except` (run time exceptions such as overflow).

An exception graph for this action is shown in Figure 5.17, again assuming that no more than two exceptions are raised concurrently. For example, if both press 1 and robot rotation motors fail simultaneously, this exception graph will be searched and the resolving exception `rs_m_failure & pr1_failure` will be raised instead of the individual exceptions `rs_m_failure` and `pr1_failure`, so that a suitable handler for this particular situation can be invoked. Any undefined exception pairs will not be resolved and will simply lead to the raising of the `universal` exception. (The handler

for the `universal` exception is responsible for stopping the system and leaving the production cell in a pre-defined safe state, if possible.)

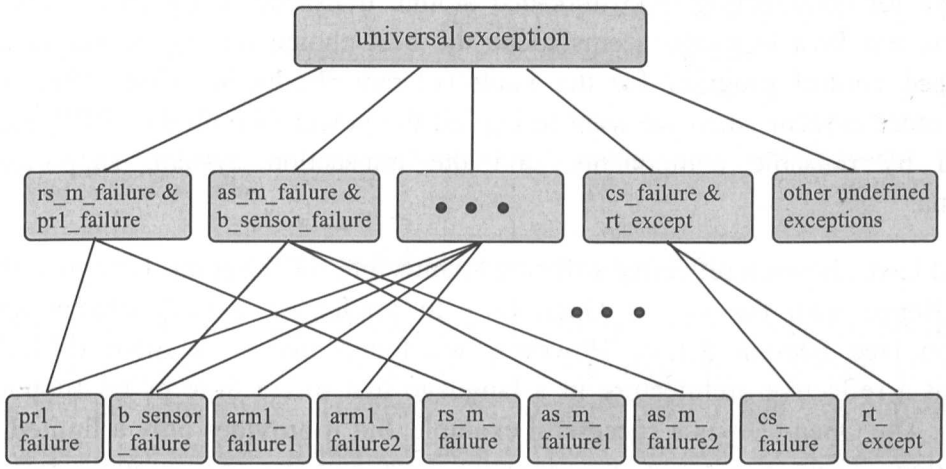


Figure 5.17 Exception graph for CA action `LoadPress1`

5.3.6 Implementation of the Control Program

In this section, we will discuss an actual implementation of a control program for the Fault-Tolerant Production Cell based on our CA action-based design in Section 5.3.3 and using the GSFT architectural pattern introduced in Section 5.2. We first explain why we choose Java as the implementation language and why we take an implementation method different from a reflective solution. We then examine to what extent the GSFT pattern helps easy the implementation of the control program and which aspects of the implementation are application-specific and thereby requiring application-dependent solutions. Finally, we describe briefly the support mechanism for dependability evaluation based on fault injection.

Implementation Language and Architecture

The Fault-Tolerant Production Cell to be controlled by a control program is actually a Tcl/Tk simulator provided by the FZI. The simulator offers a set of standard interfaces through which a control program can get various sensor values from the simulator and send various control commands to the simulator. In response to a given control command, the simulator performs the required device movement. The control program itself is independent of the simulator and can be written using any general-purpose programming language.

We would have chosen C++ or Ada 95 as the implementation language because of our previous C++ experience in implementing software fault tolerance for sequential programs (see Section 3.4) or our Ada experiment for evaluating exception handling in a distributed object system (see Section 4.3.3). However, our major experience related to CA actions has been the prototype implementation of JavaCAaction — an API for programming CA actions based on the JavaArjuna system (see Section 4.4.3). Within

JavaCAaction, a library of reusable objects that provides main services needed for implementing a CA action has been available together with the tools provided by JavaArjuna for constructing multi-threaded atomic transactions. Given our previous experience, the Java language seems to be the best choice for implementing a CA action-based control program for the Fault-Tolerant Production Cell. This choice becomes more evident when we want to exploit the power of both the GSFT pattern, supported by reusable components, and the transaction system, supported by JavaArjuna.

We would have chosen a reflective software architecture for the control program due to our experience with the use of Open C++ to implement a fault-tolerant sorting application (see Section 5.1.3). However, we found that it is often difficult to implement a reflective architecture in a language that offers little or no support for reflection. The Open C++ is a successful example, but it provides only a limited form of reflection. There are some experimental Java systems that provide reflective capabilities, but none of them has received wide acceptance. Another reason for choosing a non-reflective architecture is that we has documented the GSFT pattern in Section 5.2 as an object-oriented solution using inheritance and delegation. A similar implementation architecture will help us to examine the benefits and drawbacks of using the GSFT pattern in an actual application.

Using the GSFT Pattern to Facilitate the Implementation

The GSFT pattern for implementing software fault tolerance in a concurrent object system includes four reusable components: `external interface`, `generic FT-m-interface`, `variant` and `adjudicator` (see Figure 5.7). These components are defined respectively in the form of a Java class to facilitate the implementation of a Java-based application. The abstract `CAaction` class defined in Program 4.1 provides a standard external interface for CA actions. (It is important to notice that this abstract class is a direct implementation of the CA action specification in COALA in Section 5.3.3.) The generic FT-m-interface component is a Java class that offers a set of operations for controlling the execution of software variants and adjudicators according to a variety of fault tolerance schemes (e.g. adaptive recovery and fault masking). This interface component exercises the actual control through two abstract classes `variant` and `adjudicator`.

In order to implement the control program for the Fault-Tolerant Production Cell, we define each of 12 main CA actions as a Java class, which may be associated with several action variants and an adjudicator for the purpose of tolerating software faults. To actually program a CA action using the GSFT pattern, the initial step is to define a Java class that extends the abstract `CAaction` class. By way of example, Program 5.8 shows part of the definition of the `LoadPress1` CA action. This extended definition specifies the actual body of the action by overriding the abstract `inAction` operation in class `CAaction`. It is also responsible for declaring the shared local objects used for coordinating the roles within the action.

```
Class LoadPress1 extends CAaction
  local robotPress1Local;            //local object used by roles
  public LoadPress1() {
        robotPress1Local = new Local(this, "robotPress1Local");
        SharedLocalObject list[] = {robotPress1Local};
        … … … …
  }
  public void inAction (…) throws e { //actual body of the LoadPress1 action }
  //other related operations
}
```

Program 5.8 The `LoadPress1` action

With respect to action variants and the adjudicator, the `LoadPress1` CA action serves as a container action that is responsible for controlling the execution of both the variants and the adjudicator. However, the container action does not implement the actual control mechanism. Instead, it uses services provided by the generic FT-m-interface in the GSFT pattern. The `inAction` operation of the container action simply requests the service that implements the control using the AR scheme, and passes the reference to the action variants and the adjudicator to the interface component.

The action variants are defined by extending the abstract `variant` class (instead of being derived from the abstract `CAaction` class). Each action variant is implemented to meet the same functionality requirements but using diverse data structures and algorithms. For a CA action, an acceptance test is designed mainly based on the post-conditions of the action specified in the design phase. To implement the test, an `accepTest` class is defined by extending the abstract `adjudicator` class. The actual acceptance test is implemented as an operation that overrides the abstract `getResult()` operation in the definition of class `adjudicator`.

Following a similar and simple method, each of 12 main CA actions can be implemented as an FT-m-component and thereby equipped with the ability to tolerate software faults. The software fault tolerance scheme chosen for these actions is the AR approach. The original AR control mechanism built in the interface component provides only a basic version of the control. However, the GSFT pattern offers us the flexibility of enhancing the control mechanism by modifying the `Controller` class (see Figure 5.11). The `action` operation of class `Controller` can be re-defined using some more advanced control functions. The current implementation of the `action` operation uses the following control algorithm based on a structuring framework for exception handling:

1) execute the `inAction` operation of Variant One and then perform the `getResult()` operation of class `accepTest` — if no any exception is raised, go back to the caller;

2) if an exception is raised during the execution or `getResult()` returns `ERROR`, then invoke the corresponding handler; if no corresponding handler is found,

invoke the `generic` handler. (In either of the two exceptional cases the control is passed to the handler.)

The `generic` handler is responsible for i) restoring the `blank` object by calling its `undo` operation, ii) restoring the device objects by executing a set of reverse operations, and iii) invoking a new action variant. In the interests of simplicity and brevity, the current implementation of this handler permits just one alternate, i.e. Variant Two. Whenever an exception is raised or the acceptance test returns `ERROR`, the handler will signal an `abort` exception if the state restoration is successful, or signal a `failure` exception to the caller, i.e. an instance of the `AR` class.

Together with the CA action abstraction, the exception handling framework we used to perform error recovery allows the controlled usage of both forward and backward recovery techniques. For example, to implement forward error recovery a handler can send compensatory messages to external device objects that may have been affected by an erroneous command from the control program. This is particularly useful for our control system that has to interact with environmental objects that often cannot be simply backed up, e.g. situations in which a blank has been forged incorrectly or a blank has been dropped outside the device area.

By definition, the execution of a CA action will only produce one of the four forms of output: a normal outcome, an exceptional outcome, an `abort` exception, or a `failure` exception. Following the GSFT pattern, it is also possible that a CA action refuses performing any required computation. Instead, it signals an `interface` exception to the part of the system that made the invalid request. This type of exceptional situations may occur due to a variety of possible reasons. For example, the interface checks of the CA action detected that the pre-conditions for the execution of the CA action do not hold, or some unexpected threads attempted to participate in the performance of the CA action. We have used a `PASS`-based method for identifying the intended participants and for excluding any other unwanted threads (see Section 4.4.3).

During the process of using the GSFT pattern to implement the control program, we have leant several lessons. It is evident that the GSFT pattern enables reuse of the reference architecture for implementing software fault tolerance and simplifies to a great extent the development of a fault-tolerant object by separating different concerns. However, the pattern provides little support for addressing functional aspects of a CA action. While a pattern captures many key properties of a software architecture, it suppresses many implementation details. We feel that patterns should be treated as just one of many important tools in a toolkit of supporting software development. It is not realistic to expect that patterns free developers completely from complex analysis, design and implementation issues. An actual application like the Fault-Tolerant Production Cell often requires the use of application-specific design and implementation strategies. For example, the room for reusing an application-independent backward recovery strategy is rather severely limited in the Production Cell case study since it involves very intensive interaction between the control program

and the cell. In many cases, application-specific reverse operations are required in order to bring external device objects to a previous state.

The interaction that must be controlled within a CA action includes the cooperation between roles of the CA action, concurrent access to the external objects, and interaction between device objects and the action. We take the `blank` object as an example to demonstrate how its atomicity is maintained in our implementation. An `blank` object must be defined as an external atomic object, and can be implemented by extending the abstract `AtomicObject` class of the JavaCAaction API system. Every new class extended from this class is provided with transactional semantics, i.e. `begin`, `commit`, and `undo` operations, but must provide its own definitions of `commitState` and `undoState` operations. The actual transactional semantics of CA actions are achieved by implementing a multi-threaded CA action interface on top of the JavaArjuna system. JavaArjuna then employs nested transactions to control the state changes of atomic objects and to ensure that only consistent state transformations occur on the objects despite concurrent access and hardware-related failures such as node crashes in a distributed system.

Unlike internal objects declared by a related CA action and used for coordinating the cooperation of roles of the action, external objects are passed to CA actions via input parameters when activating a role. A corresponding transaction will be issued on the external `blank` object passed to a CA action whenever a new instance of the action starts. The transaction will end when the CA action ends. Our JavaCAaction API system implements these operations based on the multi-threaded mechanism in JavaArjuna: the thread that first arrives at the CA action interface starts a transaction, and other subsequent threads will then join the same transaction (instead of starting a new transaction). At the exit, the transaction is ended by removing all the participating threads from the transaction synchronously.

Supporting Fault Injection and Dependability Evaluation

Once the control program is implemented, it must be evaluated to determine whether the Fault-Tolerant Production Cell system meets reliability and safety objectives. Although our design of the control program has been validated using the model-checking technique in [Canver et al 1998], an analytical model such as a Markov model (see Section 3.3) should be developed to evaluate various dependability metrics for the system. Another important way to evaluate dependability is fault injection. We have used a simple method of injecting software faults directly into the control program during the implementation and testing phases of the system life cycle. Hardware device and sensor failures can be injected into the system through a failure injection panel even when the Production Cell is in operation.

Figure 5.19 shows a modified version of the failure injection panel provided originally by FZI. By using this panel various mechanics and sensor failures can be easily injected into the Production Cell simulator. For example, a rotary motor failure or a

rotary sensor failure of the robot can be injected by pressing the corresponding buttons in the panel. We have extended the original FZI panel to permit the injection of concurrent failures: a pair of failures can be injected into the simulator if the failure mode selection is set to "double". In this mode, two different failure buttons may be pressed sequentially, but only the second press will stimulate the actual injection of concurrent failures. After one or more failures are injected into the simulator, failure detection measures embedded in our control program should be able to detect them promptly. One or more corresponding exceptions will thus be raised. The simulator will then portray how such exceptions are handled within the CA action framework, in particular how the system is, if necessary, brought to a stop in a safe state.

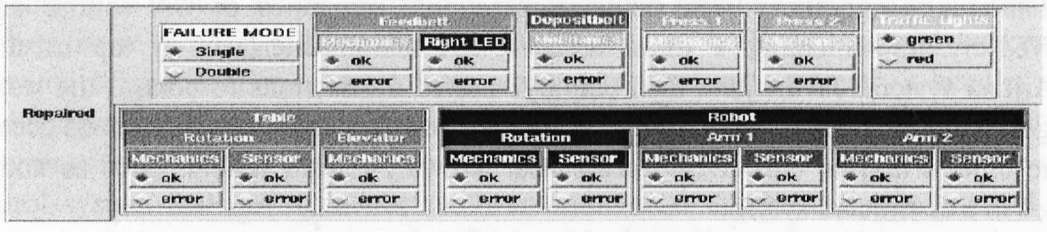


Figure 5.19 Revised failure injection panel

During the testing phase and the demonstration of our implementation, all injected device or sensor failures were caught successfully and handled immediately by the control program. This demonstrates the control program is highly robust when dealing with environmental faults. However, it is difficult to determine precisely the ability of the control program to tolerate software faults although we know that the ability depends mainly upon the error-detection coverage provided by a combination of the acceptance test, executable assertion statements and run-time checks by the hardware system. We also found that creating a set of meaningful software faults that could pass these tests and checks is not an easy task at all. Nevertheless, some events that occurred at run-time provided quite encouraging feedback. A previously unknown software fault remaining in the FZI simulator was detected successfully by the acceptance test of a CA action and recovered by the re-try operation associated with that action. It becomes evident that if a similar software error occurred inside the CA action, it would have been caught by the acceptance test and would have been tolerated successfully by a second action variant. We are now in the stage of collecting experimental data for further dependability and performance-related evaluation. We are also developing an analytical model for performing some theoretical analysis.

5.3.7 Experience and Lessons

Our experimental implementation demonstrates that the use of CA actions facilitated greatly our ability to guarantee the reliability and safety requirements of the Fault-Tolerant Production Cell. The resulting system has a clear and simple design that is easy to understand and validate.

System structuring and complexity control. Software bugs and failures of electro-mechanical components in the Fault-Tolerant Production Cell are of major concern and must be handled carefully. This requires a control program which is much more complex than the program developed for the original, non-fault-tolerant production cell. CA actions serve as a basic structuring tool to organize and design the control program. The main characteristics of our design are the way it separates safety, functionality, and efficiency concerns. In particular, the reliability and safety requirements are satisfied at the level of CA actions, while the other requirements are met by the device/sensor-controllers. The CA action structuring facilitated both the design and validation tasks by enabling the various dependability problems (involving possible clashes of moving machinery) to be treated independently of each other and of all the other aspects of the system. Even complex situations involving the concurrent occurrence of many possible software faults, mechanical failures and sensor failures could be handled simply yet appropriately.

Design for validation. Based on our design, a significant proportion of the safety, liveness, and fault-tolerance requirements for the Production Cell II case study have been formalized and model-checked in [Canver et al 1998]. The properties were expressed in terms of CTL formulae over the transition system for the CA action-based design formalized in SMV. The analysis of properties of the Fault-Tolerant Production Cell was carried out in parallel with the development of our design. Model-checking helped us to find several flaws in early versions of the design.

For example, a problem was identified that affected the order in which the robot interacts with the devices around it. This problem does not occur in the single blank instance and it is thus hard to detect by just reviewing the specification text. However, if two blanks are put into the system then the robot could manoeuvre itself into a deadlock situation from which no further activities were possible. Such "critical" sequences of actions can be derived from counter-example paths generated by the model-checker. Moreover, the counter-example also helps in finding solutions to the detected problem: we solved the problem by appropriately weakening the pre-conditions of the actions that should be executed if no deadlock occurs.

Exception handling and software fault tolerance. As a result of the experience we have gained during the process of designing and formalizing the control software, we conclude that all the dependability aspects, especially reliability and safety, of the case study can be solved very directly using the CA action mechanism, despite the need to add extensive exception handling mechanisms. The general model and mechanisms we developed in Section 4.3 provided appropriate and convenient support for handling complex exceptional situations in the Fault-Tolerant Production Cell. The schemes discussed in Section 4.4 also helped us to cope with software faults within the CA action structure. We feel that the CA action structuring together with concurrent exception handling and software fault tolerance provides an extremely powerful means of error containment and recovery, capable of dealing with very complex situations, including various concurrent failures.

System-level support and reusable components. Our previous experience of implementing fault-tolerant software for a variety of applications such as banking systems and large data sorting has greatly facilitated fast prototyping of our experimental system that controls the Fault-Tolerant Production Cell. The system-level support introduced in Sections 5.1 and 5.2 has provided a well-structured way of addressing different concerns. The reusable CA action support mechanism and components have been used quite successfully for both Production Cell I [Zorzo et al 1999] and the Fault-Tolerant Production Cell and is now being used for developing fault-tolerant control programs for the Real-Time Production Cell model [Romanovsky et al 1998].

System performance. We have concentrated on clear system structuring rather than maximizing the system performance. Performance could be improved by allowing more concurrent actions but at the cost of simplicity, maintainability and perhaps safety. What we have done is to allow as much parallelism as possible without compromising reliability and safety. CA actions themselves impose performance overheads as well, such as additional message passing and overall system synchronization. It would be possible for us to design a system with better performance by taking advantage of certain low-level knowledge of the application and by making a good trade-off between information encapsulation and parallelism.

5.4 Summary

The design and implementation of fault-tolerant software for critical computer applications are a complex and error-prone task. It needs an architectural solution that separates different concerns and makes certain aspects transparent to a given type of programmers. We have developed a multi-level reference architecture for implementing fault-tolerant software, which separates application-specific functionality, interfaces to fault tolerance schemes and application-independent control mechanisms. Such separation has helped us to promote better understanding of both functional and non-functional aspects of an application and might have resulted in increased dependability of the application. Our proposed architecture is simply based on a variety of reusable components that may be constructed according to the structuring framework of an idealized fault-tolerant component.

Our approach provides system-level support for fault-tolerant software using pre-defined classes and run-time libraries. In principle, it does not require special pre-processors or builders like the architecture introduced in [Ancona et al 1990] or a new programming language with particular syntax for specifying fault tolerance schemes. Since different groups of components are located at different levels and low-level services and implementation-details are hidden from higher-level components, our reference architecture may be ported to a number of different platforms, without requiring any direct support from a special underlying operating system. Huang and Kintala of AT&T Bell Labs [Huang & Kintala 1993] developed a library-based approach to checkpointing and backward error recovery. They introduced three

reusable components in C that provide fault tolerance in the application layer. Our solution is much more general and can be used as a unifying architecture for implementing a wide range of software fault tolerance schemes for both sequential and concurrent programs.

Although there are various solutions to the problem of supporting software fault tolerance in the application layer, it remains difficult to reuse fault-tolerant software directly for complex applications due to the growing heterogeneity of hardware and software architectures and the increasing diversity of operating system platforms. We believe design patterns are a very promising technique for achieving widespread reuse of our architectural solution. We have used the GSFT pattern to detail the reusable components in our reference architecture and to capture the static and dynamic structures and collaborations of those components. Our pattern expresses the structure and collaboration of participating components at a level still higher than source code or object-oriented design models that focus on individual objects and classes. Thus, the GSFT pattern can facilitate reuse of software architecture, even when other forms of reuse are infeasible. In general, patterns can help to reduce the development effort and make it easy to customize the core solution, but they do not specify a fully detailed solution. To find out further the design and implementation details specific to an actual application, we have conducted a case study that presents a realistic industry-oriented problem, where fault tolerance and safety requirements play a significant role.

Our case study is based on an extended production cell model that represents a manufacturing process involving redundant mechanical devices provided in order to enable continued production in the presence of faults. The challenge posed by the model specification is to design a control system that maintains specified dependability and liveness properties even in the presence of a large number and variety of software faults, device failures and sensor failures. In order to develop the required control program, we have conducted an analysis of possible software bugs and component failures and identified the various ways of detecting and handling these failures. We have used the results of this analysis to guide the design of a system employing what is in fact a very sophisticated exception handling scheme, capable of dealing appropriately even with concurrent occurrences of any of the wide variety of possible failures. We have also used the GSFT pattern to facilitate the implementation of the control system.

In light of the fact that the original (non-fault-tolerant) Production Cell was the subject of extensive studies using various formal approaches, we should emphasize that to the best of our knowledge our work represents the first and so far only complete design with formal analysis and validation [Canver et al 1998] for the much more complex and realistic Production Cell II. The work in [Matos & White 1998] describes a system design for Production Cell II that focuses just on a dynamic and transparent reconfiguration scheme that preserves safety properties. Our design is essentially different, and focuses mainly on cooperation between devices during both normal execution and the process of exception handling. A Formal Risk Analysis approach

was developed in [Liggesmeyer & Rothfelder 1998] for analyzing the run-time behaviour of Production Cell II, and studying how various sensor and actuator faults could affect both system reliability and safety. However, their analysis is not complete, and only uses the elevating rotary table of the Production Cell as an example. In contrast, our analysis is much more comprehensive and complete, including the classification of various software faults and mechanical failures and the identification of possible failures related to every device in the cell. This analysis leads further to the design of a complete control system and an actual, workable implementation.

Chapter 6

Conclusions

This thesis has developed several new techniques for building fault-tolerant software, addressed the problem of achieving fault tolerance in concurrent and distributed object systems and studied system-level support for implementing such dependable software and systems. This final chapter will summarize the major contributions of our work and give an indication of possible directions of future research.

6.1 Major Contributions

In this thesis, two advanced software fault tolerance schemes have been developed in order to increase software reliability and improve trade-offs between dependability and efficiency. For complex concurrent and distributed systems, the coordinated atomic action scheme has been examined thoroughly, together with the development of formal descriptions, concurrent exception handling and object-based diversity techniques. The problem of providing appropriate system-level support has been addressed in detail by defining a multi-level reference architecture and its associated architectural pattern. Most of the concepts and techniques developed in this work have been applied to an industrial safety-critical application and the resulting control system has been proved to be both reliable and safe through experiments and formal validation [Canver et al 1998]. To be more specific, the major contributions which have been made by this research can be summarized as follows:

- A comprehensive survey of state of the art techniques and state of practice approaches to software fault tolerance has been given with an abundant bibliography which covers latest progress. It has been used as the basis for the development of new techniques and experiments in this thesis. We believe it will be also very helpful for any project developed by other researchers in the area of software fault tolerance.

- For building sequential fault-tolerant software, two new schemes have been developed. The $t/(n-1)$-VP technique is aimed at increasing software reliability and controlling additional complexity, while the SCOP technique presents an adaptive means of dynamically adjusting software reliability and efficiency aspects. Both dependability and efficiency improvements achieved by $t/(n-1)$-VP

and SCOP have been proved analytically and supported by experimental evaluation as well.

- For constructing fault-tolerant concurrent systems, CA actions have been used as a general structuring mechanism. Extensive exception handling strategies have been developed within the CA action structuring abstraction, including models, algorithms and experiments. New object diversity techniques have been also introduced to the CA action mechanism to cope with potential software faults. Major properties of the CA action scheme have been described formally, and a prototype API system for programming CA actions, called JavaCAaction, has been implemented on top of the JavaArjuna system.

- For providing system-level support for implementing fault-tolerant software, a multi-level reference architecture with a configuration method and an architectural pattern has been proposed. Based on an object-oriented structuring method, the new pattern technique, we believe, has a significant potential and represents a very promising way to facilitate the task of implementing complex fault-tolerant software. Finally, an actual industrial case study, the Fault-Tolerant Production Cell has been used to examine and confirm most of the ideas developed in this research.

6.2 Directions for Future Research

Having summarized the work that has been presented in this thesis and shown that the aims set out in Chapter One have been satisfied, we must examine several important avenues down which further research could be directed. There are two immediate areas that are particularly related to our prototype implementation and experimental evaluation. It will be very interesting to extend the JavaCAaction API system to support more complex concurrency-control mechanisms and object recovery mechanisms and to investigate further the practical feasibility of the complex FM scheme. It will be also very useful to collect more experimental data and establish an appropriate analytical model in order to evaluate further the dependability of the control system for the Fault-Tolerant Production Cell. However, in an attempt to search for better understanding and thus better solutions at a more fundamental level, the following topics, we believe, are the important continuation of our work and merit further investigation.

6.2.1 N-Version Design versus One Good Version

The first research area of future work concerns the actual effectiveness of the multi-version approach to fault-tolerant software. For a complex critical application, we may decide to use either a single version design method with various advanced fault avoidance and fault removal techniques or the *N*-version design approach. The well known fact is: i) when the budget is so limited that each version of the *N*-version software has poor quality with very low reliability, the single version method will

produce a better system based on the same budget, and ii) when the budget is virtually or almost unlimited, and thus each version of the *N*-version software has the highest reliability that can be obtained using the best state of art techniques, the *N*-version method will achieve higher reliability than any single version design.

Figure 6.1 illustrates a brief relationship between the system reliability and the cost factors for the single version software and the *N*-version software. (The figure is for the purpose of illustration only, and the cost may be treated as a multi-dimensional factor in an actual cost model.) The grey part of the figure indicates an area where many applications belong to, but it is unclear for a given critical application which design method would achieve higher reliability.

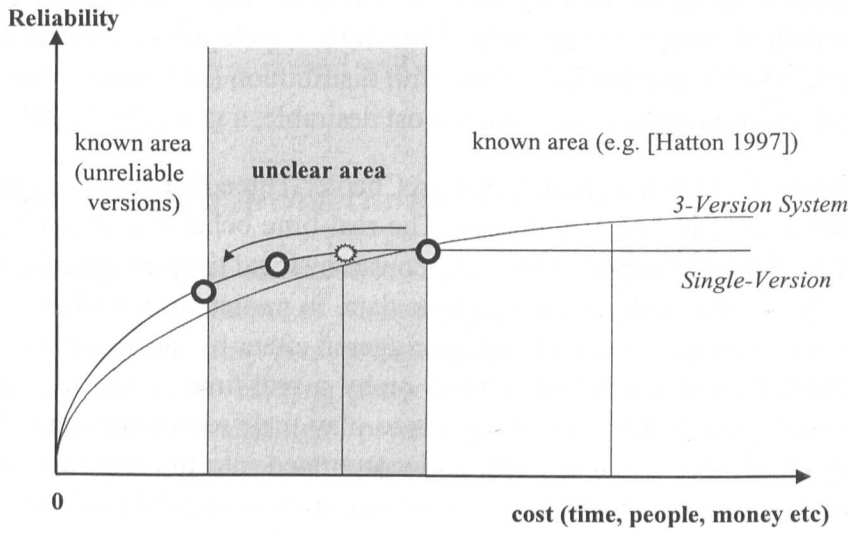


Figure 6.1 Software reliability versus cost

The real problem with which industry is often faced is whether the *limited* resources of a given critical application should be spent in producing a multi-version system, or it would be more cost-effective to spend the resources on a single version system. Unfortunately, there is no any direct answer to this question. This is one of major contributors to the decision on Boeing 777 not to use the *N*-version approach. However, when it is dangerous to rely on the *N*-version approach without strong theoretical and empirical evidence, it is also dangerous to use the single-version software in critical systems just because the approach has been used traditionally for years.

It is extremely important to seek fundamental understanding of a set of particularly difficult problems involved in such decision making. The future research should focus on the comparison of *N*-version design and one good version and the evaluation of their reliability using controlled and comparable resources. The expected results should include both experimental evaluation and analytic results, and should help industry to make decisions with an improved level of confidence.

6.2.2 Real-Time CA Actions

An important extension to the CA action mechanism is to add the support for real-time applications. From the fault tolerance viewpoint, it is important to extend the ability of CA actions to deal with exceptions in both the value and the time domain. However, this is not an easy task at all. First, exception handling in concurrent programs is still a difficult, evolving subject: no widely accepted models or approaches exist. Secondly, although most existing distributed systems are to some extent object-oriented (OO) or object-based, the object-oriented paradigm adds a new complication to system design since several aspects of this paradigm conflict with the principle of structured exception handling [Miller & Tripathi 1997]. Thirdly, real-time requirements cause further difficulties with regard to modelling the real-time behaviour of a system and to handling time-related exceptions properly. Therefore, developing a general exception handling approach that can effectively cope with distribution (and concurrency), object orientation, and real-time aspects is, though most desirable, a great challenge.

We have established a simple system model in Chapter Three that captures concepts of objects, execution threads, and CA actions. The real-time behaviour of such a system may be modelled through action-level timing constraints and time-triggered CA actions together with objects that encapsulate real-time data. In particular, it will be interesting to investigate how a given CA action can be triggered either by messages (i.e. multiple participating threads start the action jointly) or by a real-time clock that causes the automatic creation of multiple internal threads to play their respective roles. While an operation-level deadline mechanism may be incorporated into the real-time model, an extension of the conventional object model will be almost unavoidable if real-time data are to be taken into account. Attaching timing constraints to some internal data of an object becomes a natural decision since we usually assume that a CA action may not retain data. We have recently used Production Cell III, the Real-Time Production Cell model [Lötzbeyer & Mühlfeld 1996], as a new case study to deal with possibly concurrent timing and value faults. A control program for the cell is now being developed using the JavaCAaction API system. (Some initial results can be found in [Romanovsky et al 1998].)

6.2.3 Diverse Security Measures

Another interesting area of future work has become apparent during the development of our research in this thesis, which concerns security measures in large distributed systems. As a tool to guard information systems from malicious attacks, the role of software diversity should be examined. In order to achieve software fault tolerance, we have to make redundant software as diverse as possible in the hope that diverse software components do not share common design flaws. In a similar manner, the diversity approach could guard an information system effectively from malicious attacks not only by diverse software components, but also, perhaps more significantly, by diverse security measures in the sense that diverse components are not susceptible to

the same set of attacks. In other words, replaying previously successful attacks against a diverse component with a different security measure will make little sense and have little hope of success.

For a large distributed computing system under malicious attack, the real problem is again whether a single-version system is dependable, in particular secure enough to survive malicious attacks. In the situations where the cost of failure caused by attacks is extremely high and single-version systems are not secure enough, the diversity approach will be worthwhile. Some researchers may argue that redundant components are not diverse enough to survive the same attack. We believe that the whole point of the diversity approach for improving system survivability is to develop diverse components that can survive different kinds of attacks. If an attacker is not good at devising new or respective attacks on a set of diverse security measures, which can result in high cost, the entire system will survive even if some redundant components have been subverted.

6.2.4 Pattern-Oriented Architecture and Systems

The architectural pattern we discussed in Chapter Five was not supposed to be a unique solution to the development of fault-tolerant software. With the evolution of technology new patterns may evolve. The application programmers should be able to extend, modify and tailor existing patterns to their specific needs. It will be very interesting to find out how sharing of patterns could establish a common vocabulary for the design and implementation of fault-tolerant software. The development of new pattern-oriented architectures and pattern systems in this area should further ease and speed up the implementation, making fault-tolerant programs more understandable and maintainable.

6.3 In Conclusion

In this thesis, we have demonstrated a systematic approach for building fault-tolerant software, from basic concepts to object-oriented design, from sequential programs to complex concurrent systems and from architectural patterns to realistic industrial applications. As a result of the experience of we have gained during the process of developing this approach, we feel that we now have a much better understanding of software fault tolerance, including those design and implementation issues involved in realistic applications.

Any scheme for achieving software fault tolerance is based on certain form of redundancy. The additional redundancy can increase the complexity of a software system and may thereby decrease the system's dependability. The incorporation of software fault tolerance into actual systems must be performed in a disciplined and structured way. It was very pleasing to confirm from our experience that the combination of advanced fault tolerance techniques and powerful system structuring mechanisms (e.g. object-oriented structuring methods and high-level control

abstractions) often offers a quite straightforward solution to complex reliability and safety problems. System structuring aids not just the design and implementation of a software system, but also the formal verification and validation, for example by model-checking.

However, we must be clearly aware that, though contributions we made to this important area, software fault tolerance as a practical engineering discipline remains out of the reach of the average programmer and computer user. While past research in obtaining solutions for tolerating hardware faults has been much more effective, transferring advanced techniques and methodologies in software fault tolerance from an art to a routine-based practice is still a major challenge and likely to remain so for some time.

Appendixes

A Correctness of the $t/(n-1)$-VP Scheme

For the purposes of determining how "diagnosable" a given system is and performing diagnosis, a fundamental model is used. The model encompasses a representation of the system's testing assignment (who tests whom), the nature of tests and faulty units, and the implications of test results. A diagnosable system S consists of n units denoted by the set $U = \{u_1, u_2, \ldots, u_n\}$. Each $u_i \in U$ is assigned a particular subset of the remaining units in S to test. A test link, denoted by c_{ij}, corresponds to an "equality checking element" between units u_i and u_j (e.g. hardware or software comparator). The complete collection of tests in S is called the *comparison test assignment* and is represented by an undirected graph $G = (U, E)$, where each $u_i \in U$ is represented by a vertex and each edge (u_i, u_j) is in E if and only if c_{ij} is a comparison test in the comparison test assignment. A test outcome (or test result) ω_{ij} is associated with (u_i, u_j), where $\omega_{ij} = 0(1)$ if the results of a particular test task which is carried out by both units u_i and u_j agree (disagree). The collection of all outcomes is called the comparison *syndrome*. Only permanent faults are considered, even though the situation where units are intermittently faulty can be readily handled. Two classes of faults, independent faults and related faults, are further differentiated. So, two faulty units performing a same test task can compute the same incorrect results due to the manifestation of related faults.

The concept of $t/(n-1)$-diagnosability was first proposed by Friedman [Friedman 1975].

Definition A.1 [Friedman 1975]: A system S is $t/(n-1)$-fault diagnosable if and only if, given any syndrome, all faulty units can be isolated to within a set of at most $n-1$ units, provided that the number of faulty units in S does not exceed t.

It is helpful to contrast Definition 1 with the following definition of t-diagnosable systems.

Definition A.2 [Preparata et al 1967]: A system S is t-fault diagnosable if and only if, given any syndrome, all fault units can be uniquely identified, provided that the number of faulty units in S does not exceed t.

Definition A.3: For a system S and a comparison syndrome, a subset $F \subset U$ is a *consistent fault set* (CFS) if and only if 1) $|F| \leq t$; 2) $u_j \in F$ if $\omega_{ij} = 1$ and $u_i \in U - F$; 3) $u_j \in U - F$ if $\omega_{ij} = 0$ and $u_i \in U - F$.

Thus, F is a CFS for a given syndrome if and only if the assumption that the units in F are faulty and the units in $U - F$ are fault-free is consistent with the given syndrome.

Definition A.4: For a system S, a set of subsets of U, $\pi = \{V_1, V_2, \ldots, V_S\}$ where $V_i \subset U$ $(i = 1, 2, \ldots, s)$, is a cover of U if and only if $\cup_i V_i = U$ $(i = 1, 2, \ldots, s)$. A cover π is said to be *standard* if and only if $\cup_i V_i = U$ $(i = 1, 2, \ldots, s)$ and for each j $(j = 1, 2, \ldots, s)$, $\cup_i V_i \neq U$ where $i \neq j$.

With each unit $u_i \in U$ and a set of subsets of U, $\pi = \{V_1, V_2, \ldots, V_S\}$ where $V_i \subset U$ $(i = 1, 2, \ldots, s)$, we associate the sets $f(u_i) = \{V_i: V_i \in \pi \wedge u_i \in V_i\}$. For each unit $u_i \in U$ and a subset $U' \subseteq U$, we define $D(u_i) = |f(u_i)|$ and $D(U') = \Sigma D(u_i)$ where $u_i \in U'$).

To facilitate the proof of a main theorem we now introduce several simple lemmas. For t-fault diagnosable systems, Preparata, Metze, and Chien [Preparata et al 1967] gave the following necessary condition.

Lemma A.1 [Preparata et al 1967]: If a system S composed of n units is t-diagnosable, then $n \geq 2t + 1$.

Lemma A.2: If a system S composed of n units is $t/(n-1)$-diagnosable, then $n \geq 2t + 1$.

Proof: Let $G = (U, E)$ be an undirected graph representing a system S, where $|U| = n$. Suppose that S is $t/(n-1)$-diagnosable and, to the contrary, $n < 2t + 1$. Now add some undirected edges into G such that G becomes a complete graph G'. Since G' is also $t/(n-1)$-diagnosable, at least one fault-free unit can be identified. The fault-free unit can further locate all faulty units through these direct test edges between itself and the others. Therefore, G' is t-diagnosable, contracting the conclusion in Lemma A.1.

Q.E.D.

Lemma A.3: Given any syndrome, let $F_1, F_2, \ldots, F_S$ be CFS's for the syndrome. A system S represented by an undirected graph $G = (U, E)$ is $t/(n-1)$-diagnosable if and only if $U - \cup_i F_i \neq \Phi$ $(i = 1, 2, \ldots, s)$.

By Definition A.1, the statement in Lemma A.3 is immediate. Xu and Huang [Xu & Huang 1990] characterized $t/(n-1)$-diagnosable systems as follows.

Lemma A.4 [Xu & Huang 1990]: A system S represented by an undirected graph $G = (U, E)$ is $t/(n-1)$-diagnosable if and only if for any cover π of U, $\pi = \{V_1, V_2, \ldots, V_S\}$ where $|V_i| \leq t$ $(1 \leq i \leq s)$, an edge (u_i, u_j) exists such that $f(u_i) \neq f(u_j)$ and $f(u_i) \cup f(u_j) \neq \pi$.

Unlike t-fault diagnosable systems, there is no requirement on the number of units that test a unit in $t/(n-1)$-diagnosable systems. For instance, a five-unit system with four units connected as a form of "chain" and an isolated unit can be shown to be $t/(n-1)$-diagnosable ($t = 2$) by Lemma A.4 (see Figure A.1).

Lemma A.5: Let π be a cover of a unit set U. A standard cover π' of U can be produced from π.

Proof: A standard cover π' of U can be produced from π according to the following steps. Step 1: $\pi' = \pi$. Step 2: for any $V_j \in \pi'$, if $\cup_i V_i = U\ (i \neq j)$, then $\pi' = \pi' - \{V_i\}$.

Q.E.D.

Lemma A.6: Let $\pi = \{V_1, V_2, \ldots, V_S\}$ be a cover of a unit set U. There exists a subset $V_i \in \pi$ such that $|V_i| \geq D(U)/s$.

Proof: Note that $\sum_i |V_i| = \sum_j D(u_i) = D(U)$ where $i = 1, 2, \ldots, s$ and $j = 1, 2, \ldots, n$. Assume that for any $V_i \in \pi$, $|V_i| < D(U)/s$. Then, $\sum_i |V_i| < s \times D(U)/s = D(U)$, which is a contradiction.

Q.E.D.

The first class of systems we will examine is chains. A chain is formally defined below.

Definiton A.5: A chain C in an undirected graph $G = (U, E)$ is an alternating sequence of distinct vertices and edges of G, $u_1e_1u_2e_2\ldots u_{k-1}e_k$ such that for $i = 1, 2, \ldots, k-1$, $e_i = (u_i, u_{i+1})$. The set of vertices (or edges) of C is denoted by $U(C)$(or $E(C)$). $L(C) = k$ is referred to as the length of chain C.

In the following, we will fully characterize $t/(n-1)$-diagnosable chains by analysing the consistency of multiple fault sets with any given syndrome. Recall the necessary condition that $n \geq 2t + 1$. It is hoped that n would only increase linearly with the upper bound t. However, Theorem A.1 will show that this is impossible.

Lemma A.7 [Xu 1991]: For a chain C and a standard cover π of $U(C)$ where $|U(C)| = n$ and $|\pi| = s \geq 3$, if for any edge (u_i, u_j), $f(u_i) = f(u_j)$ or $f(u_i) \cup f(u_j) = \pi$ then

$$D(C) \geq n + (s-1)^2.$$

Theorem A.1: A chain C composed of n units is $t/(n-1)$-diagnosable if and only if

$$n > 2t + 1 \qquad \text{for } n \leq 8;$$

$$n > ((t+2)/2)^2 - 1 \qquad \text{for } n > 8.$$

Proof: By Lemmas A.2 and A.3, the necessity holds. By Lemmas A.4, A.5, A.6 and A.7, the sufficiency is straightforward.

Q.E.D.

From Theorem A.1, it follows that an optimal $t/(n-1)$-diagnosable chain contains at least $((t + 2)/2)^2$ units (for $n > 8$) but uses only $n-1$ comparison tests. When we perform more tests, we can certainly obtain higher diagnosability. This leads to the study of loops and $H_{2r,n}$ graphs.

Definition A.6: An undirected graph is said to be an $H_{2r,n}$ graph if there exists an edge in $H_{2r,n}$ formed by u_i ($1 \le i \le n$) and u_j if and only if $i - r \le j \le i + r$ (mod $n + 1$, $r = 1, 2, 3, ...$).

A loop is a special $H_{2r,n}$ graph with $r = 1$.

Theorem A.2: A loop $H_{2,n}$ composed of n units is $t/(n-1)$-diagnosable if and only if

$$n > ((t + 1)/2)^2.$$

Theorem A.3: An $H_{2r,n}$ system composed of n units is $t/(n-1)$-diagnosable if and only if $n \ge 2t + 1$ and

$$r \ge (t - 1)/5 \qquad \text{for } t \ge 6.$$

We omit the proofs of both Theorem A.2 and Theorem A.3 due to its similarity to Theorem 1. We can now have the following theorem which give a special class of design. This class of design can be used directly to build a practical $t/(n-1)$-VP scheme for software fault tolerance because the chains, loops and $H_{2r,n}$'s specified in the theorem have only modest requirement on the number of tests (i.e. tests by comparing the results produced by software variants).

Theorem A.4: A system S composed of n units (or software variants) is $t/(n-1)$-diagnosable if $n \ge 2t + 1$ and the assignment of result comparisons in the system S contains at least

1)	a chain of $2t$ units	for $1 \le t \le 2$;
2)	a chain of $2t + 1$ units	for $3 \le t \le 4$;
3)	an $H_{2r,n}$ structure with $r = 1$	for $5 \le t \le 6$;
4)	an $H_{2r,n}$ structure with $r \ge (t-1)/5$	for $7 \le t$.

Proof: For $t = 1$ or 2, it is straightforward to construct 1/2- and 2/4-diagnosable systems with the corresponding edges (see Figure A.1). For $t > 3$, the conclusions follow from Theorems A.1, A.2 and A.3.

Q.E.D.

Figure A.1 summarizes several examples of $t/(n-1)$-diagnosable systems derived from Theorem A.4 (i.e. Theorem 3.1 stated in Chapter Three), where vertices represent software variants and edges represent an assignment of result comparison pairs. For example, the $t/(n-1)$-diagnosable architecture for $n = 5$ and $t = 2$ only needs three

comparison pairs (or comparators) which can produce enough test results for the $t/(n-1)$-diagnostor to perform fault diagnosis (i.e. identify the faulty variants).

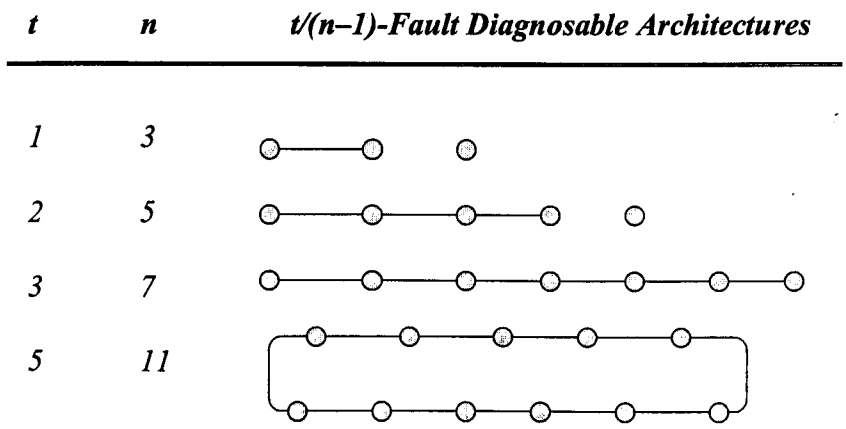


Figure A.1 Examples of $t/(n-1)$-fault diagnosable systems

B Proofs of the Correctness of Algorithm 4.1

In order to prove the correctness of our algorithm, we re-state the following assumptions.

> *Assumption B.1:* Dependable communication between threads/objects is guaranteed, i.e. no message loss or corruption.
>
> *Assumption B.2:* FIFO message passing is supported by the underlying system, i.e. two messages from thread T_i will arrive at thread T_j in the same order as they were sent.

For a specific distributed system, we assume that the following time elements can be bounded if no a fault occurs. Let T_{mmax} be the maximum time of message passing between two concurrent execution threads in the system; T_{reso} be the upper bound of the time spent in resolving exceptions, T_{abort} be the maximum possible time for a thread to abort one nested CA action, n_{max} be the maximum number of nesting levels of CA actions (if no nesting, then $n_{max} = 0$), and Δ_{max} be maximum possible time of handling a (resolving) exception. We now show that no deadlock is possible in our proposed algorithm.

Lemma B.1: Consider N execution threads that interact within nested CA actions. For any thread T_i, if it reaches the state X (exceptional) or S (suspended), it will complete exception handling ultimately in at most $\boldsymbol{T}$, where

$$\boldsymbol{T} \leq (2n_{max} + 3)T_{mmax} + n_{max}T_{abort} + (n_{max} + 1)(T_{reso} + \Delta_{max}).$$

Proof: In order to prove the above bound, let us consider the worst case, i.e. a thread that raises an exception is in the innermost CA action and each time the abortion of a nested action occurs right at the end of exception handling within that nested action.

Without loss of generality, assume that a thread T_i in the innermost action raises an exception and changes its state into X. It will send the exception message to all the other participating threads, by assumption 1, which will reach them in T_{mmax}. Since there are no further nested actions within the innermost action, any message from the other threads about an exception or suspended state will come to T_i in at most $2T_{mmax}$. Note that actual exception resolution may take T_{reso}. Therefore, T_i will receive a resolving exception and then complete exception handling in at most $(3T_{mmax} + T_{reso} + \Delta_{max})$.

If T_i has not yet left the innermost action, but a further exception occurs in its direct containing action, then the abortion of the innermost action will have to be performed. After the abortion, T_i will send either an abortion exception or suspended message to other threads, which will arrive at them in $(T_{abort} + T_{mmax})$. T_i will then receive the resolving exception (or resolve the exceptions by itself) in at most $(T_{reso} + T_{mmax})$ and complete exception handling within Δ_{max}. The whole process costs at most $(2T_{mmax} + T_{abort} + T_{reso} + \Delta_{max})$.

In the worst case, the above process could be repeated n_{max} times until the outermost CA action is reached. Totally the repeated process will cost at most $n_{max}(2T_{mmax} + T_{abort} + T_{reso} + \Delta_{max})$. Adding the time spent in the innermost action, we therefore have that

$$T \leq (2n_{max} + 3)T_{mmax} + n_{max}T_{abort} + (n_{max} + 1)(T_{reso} + \Delta_{max})$$

namely, thread T_i will complete exception handling ultimately and leave the outermost CA action.

Q.E.D.

By Lemma B.1, we know that any thread will complete exception handling within a finite time bound. Therefore, deadlock during the process of exception handling will be impossible while executing the proposed algorithm. However, in order to prove the entire correctness of the proposed algorithm, we must show that any resolving exception is a proper cover of the multiple exceptions that have been raised concurrently so far.

Lemma B.2: For a given CA action A, if no exception is raised in any containing action of A, then no more new exceptions will be raised within A once the exception resolution starts.

Proof: Assume that, to the contrary, a new exception message arrives at the resolving thread after it has started the resolution. Note that, from the proposed algorithm, the resolving thread must know all the states (X or S) of the participating threads in A before it can begin any actual resolution. Hence, by assumption B.2, the only possibility is that the newly arriving exception is caused by an abortion event, namely,

A must be aborted by some containing action, contradicting the assumption that no exception is raised in any containing action of A.

Q.E.D.

Lemma B.3: Consider N execution threads that interact within nested CA actions. If multiple exceptions are raised concurrently, an ultimate resolving exception that covers all the exceptions will be generated by the proposed algorithm.

Proof: An exception that is raised in the containing CA action will abort any effect the nested action may have made or be making (even if a resolving exception for the nested action has been identified and the corresponding exception handling has been in operation). Note however that the number of nesting levels is finite and bounded by n_{max}. Abortion will be no longer possible if the current active action A is the outermost (or top-level) CA action. By Lemma B.2, the exception resolution will start finally and no more new exception will be raised.

Q.E.D.

From Lemmas B.2 and B.3, we know that a resolving exception produced by Algorithm 4.1 will always cover all the exceptions raised concurrently so far. Any further exception will cause the abortion of any effect of previous resolutions and trigger a new exception resolution. Because deadlock is not possible, within a limited amount of time a final resolving exception will be raised in the end. We therefore have the conclusion (i.e. Theorem 4.1 in Chapter Four) below.

Theorem B.1: The proposed algorithm is deadlock-free and always performs correct exception resolution.

C Communication Complexity of Algorithm 4.1

Without the nesting of CA actions, it is obvious that the message complexity of our algorithm is $O(N^2)$ messages, where N is the number of the threads participating in the outermost CA action. More precisely,

1) when only one exception is raised and there are no nested actions, then the number of messages is $(N + 1) \times (N - 1)$, i.e. $(N - 1)$ `Exception`, $(N - 1)^2$ `Suspended`, and $(N - 1)$ `Commit` messages;

2) when all N participating threads have the exceptions raised simultaneously, the number of messages is also $(N + 1) \times (N - 1)$, i.e. $N \times (N - 1)$ `Exception` and $(N - 1)$ `Commit` messages.

From the proposed algorithm, we can see that the number of messages is in fact independent of the number of concurrent exceptions, which is a great improvement over our previous algorithm in [Romanovsky et al 1996]. Taking the nesting of actions into account, we have the theorem (i.e. Theorem 4.2 in Chapter Four) below.

Theorem C.2: In the worst case, our proposed algorithm requires exactly $n_{max} \times (N^2 - 1)$ messages.

References

[Agha et al 1992] G. Agha, S. Frolund, R. Panwar, and D. Sturman, "A linguistic framework for dynamic composition of dependability protocols," in *3rd Int. Conf. Dependable Comput. Critical Appli.*, pp.197-207, Mondello, 1992.

[Ammann & Knight 1988] P.E. Ammann and J.C. Knight, "Data diversity: an approach to software fault tolerance," *IEEE Trans. Comput.*, vol. 37, no. 4, pp.418-425, 1988.

[Ancona et al 1990] M. Ancona, G. Dodero, V. Gianuzzi, A. Clematis, and E.B. Fernandez, "A system architecture for fault tolerance in concurrent software," *IEEE Comput.*, vol. 23, no. 10, pp.23-32, 1990.

[Anderson & Kerr 1976] T. Anderson and R. Kerr, "Recovery blocks in action: a system supporting high reliability," in *2nd Int. Conf. on Soft. Eng.*, pp.447-457, San Francisco, 1976.

[Anderson & Knight 1983] T. Anderson and J.C. Knight, "A framework for software fault tolerance in real-time systems," *IEEE Trans. Soft. Eng.*, vol. SE-9, no. 3, pp.355-364, 1983.

[Anderson & Lee 1981] T. Anderson and P.A. Lee. *Fault Tolerance: principles and practice,* Prentice-Hall, 1981.

[Anderson 1986] T. Anderson, "A structured decision mechanism for diverse software," in *5th Symp. Reliability Distrib. Soft. & Database Syst.*, pp.125-129, Los Angeles, 1986.

[Anderson et al 1985] T. Anderson, P.A. Barrett, D.N. Halliwell and M.R. Moulding, "Software fault tolerance: an evaluation," *IEEE Trans. Soft. Eng.*, vol. SE-11, no. 12, pp.1502-1510, 1985.

[ANSI 1991] ANSI/IEEE, *Standard Glossary of Software Engineering Terminology*, STD-729-1991, ANSI/IEEE, 1991.

[Arlat et al 1988] J. Arlat, K. Kanoun and J.C. Laprie, "Dependability evaluation of software fault tolerance," in *18th Int. Symp. Fault-Tolerant Comput.*, pp.142-147, Tokyo, 1988.

[Arlat et al 1990] J. Arlat, K. Kanoun and J.C. Laprie, "Dependability modelling and evaluation of software fault tolerant systems," *IEEE Trans. Comput.*, vol. 39, no. 4, pp.504-513, 1990.

[Athavale 1989] A. Athavale, *Performance Evaluation of Hybrid Voting Schemes*, M.Sc thesis, Dept. of Comput. Sci., North Carolina State Univ., 1989.

[Avizienis & Ball 1987] A. Avizienis and D.E. Ball, "On the achievement of a highly dependable and fault-tolerant air traffic control system," *IEEE Comput.*, vol. 20, no. 2, pp.84-90, 1987.

[Avizienis & Chen 1977] A. Avizienis and L. Chen, "On the implementation of *N*-version-programming for software fault-tolerance during execution," in *Int. Conf. Comput. Soft. & Appli.*, pp.149-155, New York, 1977.

[Avizienis & Kelly 1984] A. Avizienis and J. Kelly, "Fault tolerance by design diversity: concepts and experiments," *IEEE Comput.*, vol. 17, no. 8, pp.67-80, 1984.

[Avizienis & Laprie 1986] A. Avizienis and J.C. Laprie, "Dependable computing: from concepts to design diversity," *Proc. IEEE*, vol. 74, no. 5, pp.629-638, 1986.

[Avizienis 1975] A. Avizienis, "Fault tolerance and fault intolerance: complementary approaches to reliable computing," in *1975 Int. Conf. Reliable Soft.*, pp.458-464, 1975.

[Avizienis 1985] A. Avizienis, "The *N*-version approach to fault-tolerant software," *IEEE Trans. Soft. Eng.*, vol. SE-11, no. 12, pp.1491-1501, 1985.

[Avizienis et al 1985] A. Avizienis, P. Gunningberg, J. Kelly, L. Strigini, P.J. Traverse, K.S. Tso and U. Voges, "The UCLA DEDIX system: a distributed testbed for multiple-version software," in *15th Int. Symp. Fault-Tolerant Comput.*, pp.126-134, Ann Arbor, 1985.

[Avizienis et al 1988a] A. Avizienis, M.R. Lyu, W. Schuetz, K.S. Tso and U. Voges, "DEDIX 87 - a supervisory system for design diversity experiments at UCLA," in [Voges 1987], pp.129-168, 1988.

[Avizienis et al 1988b] A. Avizienis, M.R. Lyu and W. Schuetz, "In search of effective diversity: a six-language study of fault-tolerant flight control software," in *18th Int. Symp. Fault-Tolerant Comput.*, pp.15-22, Tokyo, 1988.

[Babaoglu 1987] O. Babaoglu, "On the reliability of consensus-based fault-tolerant distributed computing systems," *ACM Trans. Comput. Syst.*, vol. 5, pp.394-416, 1987.

[Banâtre et al 1986] J.P. Banâtre, M. Banâtre and F. Ployette, "The concept of multi-functions: a general structuring tool for distributed operating systems," in *6th Int. Conf. Distrib. Comput. Syst.*, pp.478-485, 1986.

[Barborak et al 1993] M. Barborak, M. Malek and A. Dahbura, "The consensus problem in fault-tolerant computing," *ACM Comput. Surveys*, vol. 25, pp.170-220, 1993.

[Barigazzi & Strigini 1983] G. Barigazzi and L. Strigini, "Application-transparent setting of recovery points," in *13th Int. Symp. Fault -Tolerant Comput.*, pp.48-55, Milano, 1983.

[Barrett & Speirs 1993] P.A. Barrett and N.A. Speirs, "Towards an integrated approach to fault tolerance in Delta-4," *Distrib. Syst. Eng.*, vol. 1, no. 1, pp.59-66, 1993.

[Beder & Rubira 1998] D.M. Beder and C.M.F. Rubira, "A reflective object-oriented framework for developing dependable distributed software based on patterns and Metapatterns," in *28th Int. Symp. Fault-Tolerant Comput.*, FastAbstracts, pp.45-46, Munich, 1998.

[Belli & Jedrzejowiez 1991] E. Belli and P. Jedrzejowiez, "Comparative analysis of concurrent fault-tolerance techniques for real-time applications," in *2nd Int. Symp. Soft. Reliability Eng.*, Austin, 1991.

[Benveniste & Issarny 1992] M. Benveniste and V. Issarny. *Concurrent Programming Notations in the Object-Oriented Language Arche,* Research Report, no. 1822, Rennes, France, INRIA, 1992.

[Bernstein & Lewis 1993] A. Bernstein and P. Lewis, *Concurrency in Programming and Database Systems*, Jones and Bartlett Publ., 1993.

[Birman 1993] K. Birman, "The process group approach to reliable computing," *Communications of the ACM*, vol. 36, no. 12, pp.37-53, 1993.

[Bishop & Pullen 1989] P.G. Bishop and F.D. Pullen, "Error masking: a source of failure dependency in multi-version programs," in *1st Int. Dependable Comput. Critical Appli.*, pp.25-32, Santa Barbara, 1989.

[Bishop et al 1986] P.G. Bishop, D.G. Esp, M. Barnes, P. Humphreys, G. Dahl and J. Lahti, "PODS — A project on diverse software," *IEEE Trans. Soft. Eng.*, vol. SE-12, no. 9, pp.929-940, 1986.

[Bondavalli & Simoncini 1992] A. Bondavalli and L. Simoncini, "Structured software fault tolerance with BSM," in *3rd Workshop Future Trends Distrib. Comput. Syst.*, Taipei, 1992.

[Bondavalli et al 1993] A. Bondavalli, J. Stankovic and L. Strigini, "Adaptable fault tolerance for real-time systems, " in *3rd Workshop Responsive Comput. Syst.*, pp.123-132, New Hampshire, 1993.

[Brick et al 1984] D.B. Brick, J.S. Draper and H.J. Caulfield, "Computers in the military and space sciences," *IEEE Comput.*, vol. 17, no. 10, pp.250-262, 1984.

[Brilliant et al 1990] S.S. Brilliant, J.C. Knight and N.G. Leveson, "Analysis of faults in an *N*-version software experiment," *IEEE Trans. Soft. Eng.*, vol. SE-16, no. 2, pp.238-247, 1990.

[Buschmann 1995] F. Buschmann, "The master-slave pattern," in *Pattern Languages of Program Design*, (eds. J. Coplien & D.C. Schnidt), Addison-Wesley, pp.133-142, 1995.

[Buschmann et al 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: pattern-oriented software architecture*, John Wiley & Sons, 1996.

[Butler & Finelli 1993] R.W. Butler and G.B. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," *IEEE Trans. Soft. Eng.*, vol. SE-19, no. 1, pp.3-12, 1993.

[Campbell & Randell 1986] R.H. Campbell and B. Randell, "Error recovery in asynchronous systems," *IEEE Trans. Soft. Eng.*, vol. SE-12, no. 8, pp.811-826, 1986.

[Campbell et al 1979] R.H. Campbell, K.H. Horton and G.G. Belford, "Simulations of a fault-tolerant deadline mechanism," in *9th Int. Symp. Fault-Tolerant Comput.*, pp.95-101, Madison, 1979.

[Canver et al 1998] E. Canver, D. Schwier, A. Romanovsky and J. Xu, "Formal verification of CAA-based designs: the Fault-Tolerant Production Cell," 3rd Year Report, *ESPRIT Project 20072 Design for Validation*, pp.229-258, 1998.

[Carpenter & Tyrrell 1991] G.F. Carpenter and A.M. Tyrrell, "Software fault tolerance in concurrent systems: conversation placement using CSP," *Microprocessing & Microprogramming*, vol. 32, pp.373-380, 1991.

[Chen & Avizienis 1978] L. Chen and A. Avizienis, "*N*-version programming: a fault-tolerant approach to reliability of software operation," in *8th Int. Symp. Fault-Tolerant Comput.*, pp.3-9, Toulouse, 1978.

[Chen & Dayal 1996] Q. Chen and U. Dayal, "A transactional nested process management system," in *12th Int. Conf. Data Eng.*, pp.566-573, New Orleans, 1996.

[Chiba & Masuda 1993] S. Chiba and T. Masuda, "Designing an extensible distributed language with a meta-level architecture," in *ECOOP'93*, pp.482-501, 1993.

[Chrysanthis 1993] P.K. Chrysanthis, "Transaction processing in a mobile computing environment," in *IEEE Workshop APDS*, pp.77-82, 1993.

[Clematis & Gianuzzi 1993] A. Clematis and V. Gianuzzi, "Structuring conversation in operation/procedure-oriented programming languages," *Comput. Lang.*, vol. 18, no. 3, pp.153-168, 1993.

[Condor & Hinton 1988] A.E. Condor and G.J. Hinton, "Fault-tolerant and fail-safe design of CANDU computerized shutdown systems," in *IAEA Meeting Microprocessors Important to the Safety of Nuclear Power Plants*, London, 1988.

[Cristian 1982] F. Cristian, "Exception handling and software fault tolerance," *IEEE Trans. Comput.*, vol. C-31, no. 6, pp.531-540, 1982.

[Cristian 1984] F. Cristian, "Correct and robust programs," *IEEE Trans. Soft. Eng.*, vol. SE-10, no. 2, pp.163-167, 1984.

[Cristian 1995] F. Cristian, "Exception handling and tolerance of software faults," in *Software Fault Tolerance*, (ed. M.R. Lyu), John Wiley & Sons, pp.81-107, 1995.

[Csenki 1989] A. Csenski, "Recovery block reliability analysis with failure clustering," in *1st Int. Conf. Dependable Comput. Critical Appli.*, pp.33-42, Santa Barbara, 1989.

[Dahll & Lathi 1979] G. Dahll and J. Lathi, "An investigation of methods for production and verification of highly reliable software," in *SAFECOMP'79*, pp.89-94, 1979.

[Daniels et al 1997] F. Daniels, K. Kim and M.A. Vouk, "The reliable hybrid pattern: a generalized software fault tolerant design pattern," in *Int. Conf. PloP'97*, pp.1-9, 1997.

[Dasgupta et al 1988] P. Dagupta, R. Leblanc Jr. and W. Appelbe, "The Clouds distributed operating system," in *8th Int. Conf. Distrib. Comput. Syst.*, San Jose, 1988.

[Davies 1973] C.T. Davies, "Recovery semantics for a DB/DC system," in *ACM Nat. Conf*, pp.136-141, 1973.

[Davies 1978] C.T. Davies, "Data processing spheres of control," *IBM Syst. Journal*, vol. 17, no. 2, pp.179-198, 1978.

[Davies 1984] P.A. Davies, "The latest developments in automatic train control," in *Int. Conf. Railway Safety Control & Automation Towards 21st Century*, pp.272-279, London, 1984.

[Davis et al 1993] G.J. Davis, M.R. Earls and F.A. Patterson-Hine, "Reliability analysis of the X-29A flight control system software," *Journal of Comput. & Soft. Eng.*, vol. 1, no. 4, pp.325-348, 1993.

[Di Giandomenico & Strigini 1990] F. Di Giandomenico and L. Strigini, "Adjudicators for diverse redundant Components," in *9th Symp. Reli. Distrib. Syst. (SRDS-9)*, pp.114-123, Alabama, 1990.

[Di Giandomenico et al 1997] F. Di Giandomenico, A. Bondavalli, J. Xu, and S. Chiaradonna, "Hardware and software fault tolerance: definition and evaluation of adaptive architectures in a distributed computing environment," in *Advances in Safety and Reliability*, (ed. C.G. Soares), Pergamon Press, pp.341-348, June 1997.

[Dijkstra 1968] E.D. Dijkstra, "Co-operating sequential processes," in *Programming Languages*, (ed. F. Genuys), Academic Press, New York, 1968.

[Dugan & Lyu 1995] J.B. Dugan and M.R. Lyu, "Dependability modeling for fault-tolerant software and systems," *IEEE Trans. Reliability,* no. 12, pp.513-519, 1994.

[Dugan 1994] J.B. Dugan, "System-level reliability and sensitivity analysis for three fault-tolerant system architectures," in *4th Int. Conf. Dependable Comput. Critical Appli.*, pp.295-307, San Diego, 1994.

[Eckhardt & Lee 1985] D.E. Eckhardt and L.D. Lee, "A theoretical basis for the analysis of multi-version software subject to coincident errors," *IEEE Trans. Soft. Eng.*, vol. SE-11, no. 12, pp.1511-1517, 1985.

[Eckhardt et al 1991] D.E. Eckhardt, A.K. Caglayan, J.C. Knight, L.D. Lee, D.F. McAllister, M.A. Vouk, and J.P.J. Kelly, "An experimental evaluation of software redundancy as a strategy for improving reliability," *IEEE Trans. Soft. Eng.*, vol. SE-17, no. 7, pp.692-702, 1991.

[Elmagarmid 1993] A.K. Elmagarmid (ed.). *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publ., California, 1993.

[Elmendorf 1972] W.R. Elmendorf, "Fault-tolerant programming," in *2nd Int. Symp. Fault-Tolerant Comput.*, pp.79-83, Newton, 1972.

[Eppinger et al 1991] J.L. Eppinger, L.B. Mummert and A.Z. Spector. *Camelot and Avalon: a distributed transaction facility*, Morgan Kaufmann Publ., California, 1991.

[Erb 1989] A. Erb, "Safety measures of the electronic interlocking system ELEKTRA," in *IFAC SAFECOMP'89*, pp.49-52, Vienna, 1989.

[Eswaran et al 1976] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The notions of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, no. 11, pp.624-633, 1976.

[Fabre & Pérennou 1998] J-C. Fabre and T. Pérennou, "A metaobject architecture for fault-tolerant distributed systems: the FRIENDS approach," *IEEE Trans. Comput., Special Issue of Dependability of Computing Systems*, pp.78-95, 1998.

[Fabre et al 1995] J-C. Fabre, V. Nicomette, T. Pérennou, R. Stroud, and Z. Wu, "Implementing fault-tolerant applications using reflective object-oriented programming," in *25th Int. Symp. Fault-Tolerant Comput.*, pp.489-498, Pasadena, 1995.

[Ferreira & Rubira 1998] L.L. Ferreira and C.M.F. Rubira, "Integration of fault tolerance techniques: a system of patterns to cope with hardware, software and environmental fault tolerance," in *28th Int. Symp. Fault-Tolerant Comput.*, FastAbstracts, pp.25-26, Munich, 1998.

[Fischler et al 1975] M.A. Fischler, O. Firschein and D.L. Drew, "Distinct software: an approach to reliable computing," in *2nd USA-Japan Comput.*, pp.573-579, Tokyo, 1975.

[Francez & Forman 1996] N. Francez and I.R. Forman. *Interacting Processes: a multiparty approach to coordinated distributed programming*, Addison-Wesley, 1996.

[Friedman 1975] A. Friedman, "A new measure of digital system diagnosis," in *5th Int. Symp. Fault-Tolerant Comput.*, pp.167-170, Paris, 1975.

[Frullini & Lazzari 1984] R. Frullini and A. Lazzari, "Use of microprocessor in fail-safe on board equipment," in *Int. Conf. Railway Safety Control & Automation Towards 21st Century*, pp.292-299, London, 1984.

[Gamma et al 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Objected-Oriented Software*, Addison Wesley, 1995.

[Gantenbein et al 1991] R.E. Gantenbein, S.Y. Shin and J.R. Cowles, "Evaluation of combined approaches to distributed software-based fault tolerance," in *Pacific Rim Int. Symp. Fault-Tolerant Comput.*, pp.70-75, Michigan, 1991.

[Giloth & Prantzen 1983] F.K. Giloth and K.D. Prantzen, "Can the reliability of digital telecommunication switching systems be predicted and measured?," in *13th Int. Symp. Fault -Tolerant Comput.*, pp.392-397, Milano, 1983.

[Gmeiner & Voges 1979] L. Gmeiner and U. Voges, "Software diversity in reactor protection systems: an experiment," in *IFAC SAFECOMP'79*, pp.75-79, 1991.

[Gopal & Griffeth 1991] G. Gopal and N.D. Griffeth, "Software fault tolerance in telecommunications systems," *ACM Operating Syst. Review*, vol. 25, no. 2, pp.112-116, 1991.

[Grady 1992] R. B. Grady. *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, 1992.

[Gray & Reuter 1992] J. Gray and A. Reuter. *Transaction Processing: concepts and techniques*, Morgan Kaufmann, 1992.

[Gray & Siewiorek 1991] J. Gray and D.P. Siewiorek, "High-availability computer systems," *IEEE Comput.*, vol. 24, no. 9, pp.39-48, 1991.

[Gray 1978] J.N. Gray, "Notes on database operating systems," in *Operating Systems: An Advanced Course*, (R. Bayer et al eds), Springer-Verlag, pp.393-481, 1978.

[Gray 1986] J.N. Gary, "Why do computers stop and what can we do about it?," in *5th Symp. Reli. Distrib. Syst. & Database Syst.*, pp.3-12, Los Angeles, 1985.

[Gray 1990] J.N. Gray, "A census of Tandem system availability between 1985 and 1990," *IEEE Trans. Reliability*, vol. 39, no. 4, pp.409-418, 1990.

[Gregory & Knight 1985] S.T. Gregory and J.C. Knight, "A new linguistic approach to backward error recovery," in *15th Int. Symp. Fault-Tolerant Comput.*, pp.404-409, Michigan, 1985.

[Gregory & Knight 1989] S.T. Gregory and J.C. Knight, "On the provision of backward error recovery in production programming languages," in *19th Int. Symp. Fault-Tolerant Comput.*, pp.506-511, Chicago, 1989.

[Gregory 1987] S.T. Gregory. *Programming Language Facilities for Backward Error Recovery in Real-Time Systems*. PhD Dissertation, Dept. of Comput. Sci., Univ. of Virginia, 1987.

[Grnarov et al 1980] A. Grnarov, J. Arlat and A. Avizienis, "On the performance of software fault tolerance strategies," in *10th Int. Symp. Fault-Tolerant Comput.*, pp.251-253, Kyoto, Japan, 1980.

[Hachiga et al 1993] A. Hachiga, K. Akita and Y. Hasegawa, "The design concepts and operational results of fault-tolerant computer systems for the Shinkansen train control," in *23th Int. Symp. Fault-Tolerant Comput.*, pp.78-87, Toulouse, 1993.

[Haerder & Reuter 1983] T. Haerder and A, Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surveys*, vol. 13, no. 2, pp.287-5317, 1983.

[Hagelin 1987] G. Hagelin, "ERICSSON Safety system for railway control," in *Dependable Computing and Fault-Tolerant Systems*, (eds. U. Voges et al), Springer-Verlag, pp.11-27, 1987.

[Hatton 1997] L. Hatton, "*N*-version design versus one good version," *IEEE Software*, vol. 14, no. 6, pp.71-76, 1997.

[Haugk et al 1985] G. Haugk, F.M. Lax, R.D. Rover and J.R. Williams, "The 5 ESS switching system: maintenance capabilities," *AT&T Technical Journal*, vol. 64, no. 6, pp.1385-1416, 1985.

[Hayes-Roth 1995] Hayes-Roth, "A domain specific software architecture for adaptive intelligent systems," *IEEE Trans. Soft. Eng.*, vol. SE-21, no. 4, pp.288-301, 1995.

[Hecht & Hecht 1986] H. Hecht and M. Hecht, "Software reliability in the system context," *IEEE Trans. Soft. Eng.*, vol. SE-12, no. 1, pp.51-58, 1986.

[Hecht & Hecht 1996] H. Hecht and M. Hecht, "Fault tolerance in software," in *Fault-Tolerant Computer System Design*, (ed. D.K Pradhan), Prentice-Hall, pp.428-477, 1996.

[Hecht 1976] H. Hecht, "Fault-tolerant software for real-time applications," *ACM Comput. Surveys*, vol. 8, no. 4, pp.391-407, 1976.

[Hecht et al 1989] M. Hecht, J. Agron and S. Hochhauser, "A distributed fault tolerant architecture for nuclear reactor control and safety functions," in *Real-Time Syst. Symp.*, pp.214-221, Santa Monica, 1989.

[Hecht et al 1991] M. Hecht, J. Agron, H. Hecht, and K.H. Kim, "A distributed fault-tolerant architecture for nuclear reactor and other critical computing," in *21st Int. Symp. Fault-Tolerant Comput.*, pp.462-469, Montreal, 1991.

[Hennebert & Guiho 1993] C. Hennebert and G. Guiho, "SACEM: a fault-tolerant system for train speed control," in *23th Int. Symp. Fault-Tolerant Comput.*, pp.624-628, Toulouse, 1995.

[Herlihy & Wing 1987] M.P. Herlihy and J.M. Wing, "Avalon: language support for reliable distributed systems," in *17th Int. Symp. Fault-Tolerant Comput.*, pp.89-95, Pittsburgh, 1987.

[Hills 1983] A.D. Hills, "A310 slat and flap control system management and experience," in *Proceedings of 5th DASC,* 1983.

[Hoare 1978] C.A.R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp.666-677, 1978.

[Horning et al 1974] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith and B. Randell, "A program structure for error detection and recovery," *Lecture Notes in Computer Science*, vol. 16, pp.177-193, 1974.

[Huang & Kintala 1993] Y. Huang and C.M.R. Kintala, "Software implemented fault tolerance: Technologies and experience," in *23rd Int. Symp. Fault Tolerant Comput.*, pp. 2-9, Toulouse, 1993.

[Huang & Kintala 1995] Y. Huang and C. Kintala, "Software fault tolerance in the application layer," in *Software Fault Tolerance*, (ed. M.R. Lyu), John Wiley & Sons, pp.232-248, 1995.

[Issarny 1993a] V. Issarny, "An exception handling mechanism for parallel object-oriented programming: towards reusable, robust distributed software," *Journal Object-Oriented Prog.*, vol. 6, no. 6, pp.29-40, 1993.

[Issarny 1993b] V. Issarny, "Programming notations for expressing error recovery in a distributed object-oriented language," in *1st Broadcast Open Workshop,* pp.1-19, Newcastle, 1993.

[Jalote & Campbell 1984] P. Jalote and R.H. Campbell, "Fault tolerance using communicating sequential processes," in *14th Int. Symp. Fault-Tolerant Comput.*, pp.347-352, Florida, 1984.

[Jalote & Campbell 1986] P. Jalote and R.H. Campbell, "Atomic actions for software fault tolerance using CSP," *IEEE Trans. Soft. Eng.*, vol. SE-12, no. 1, pp.59-68, 1986.

[Jalote 1986] P. Jalote, "Using broadcast for multiprocess recovery," in *6th Int. Conf. Distrib. Comput. Syst.*, pp.582-589, 1986.

[Jung & Smolka 1996] Y-J. Joung and S. A. Smolka, "A comprehensive study of the complexity of multiparty interaction," *Journal of the ACM*, vol. 43, no. 1, pp.75-115, 1996.

[Kanekawa et al 1998] N. Kanekawa, T. Meguro, K. Isono, Y. Shima, N. Miyazaki, and S. Yamaguchi, "Fault detection and recovery coverage improvement by clock synchronized duplicated systems with optimal time diversity," in *28th Int. Symp. Fault-Tolerant Comput.*, pp.196-200, Munich, 1998.

[Kanoun et al 1993] K. Kanoun, M. Kaaniche, C. Beounes, J.C Laprie, and J. Arlat, "Reliability Growth of fault-tolerant software," *IEEE Trans. Reliability*, vol.42 , no. 2, pp.205-219, 1993.

[Kantz & Koza 1995] H. Kantz and C. Koza, "The ELEKTRA railway signalling-system: field experience with an actively replicated system with diversity," in *25th Int. Symp. Fault-Tolerant Comput.*, pp.453-458, Pasadena, 1995.

[Kaplan 1985] G. Kaplan, "The X-25: Is it coming or Going?," *IEEE Spectrum*, vol. 22 , no. 6, pp.54-60, 1985.

[Kelly & Avizienis 1983] J.P.J. Kelly and A. Avizienis, "A specification-oriented multi-version software experiment," in *13th Int. Symp. Fault-Tolerant Comput.*, pp.120-126, Milano, 1983.

[Kelly et al 1986] J.P.J. Kelly, A, Avizienis, B.T. Miller, R.T. Lyu, A.T. Tai, and K.S. Tso, "Multiversion software development," in *IFAC SAFECOMP'86,* pp.43-49, 1986.

[Kelly et al 1988] J.P.J. Kelly, D. Eckhardt, A. Caglayan, J. Knight, D. McAllister, and M.Vouk, "A large scale second generation experiment in multi-version software: description and early results," in *18th Int. Fault-Tolerant Comput.* , pp.9-14, 1988.

[Kiczales et al 1991] G. Kiczales, J. des Rivieres and D.G. Bobrow. *The Art of the Metaobject Protocol,* MIT Press, 1991.

[Kim & Min 1991] K.H. Kim and B.J. Min, "Approaches to implementation of multiple DRB stations in tightly coupled computer networks and an experimental validation," in *15th Int. Conf. Comput. Soft. & Appli.*, pp.550-557, Tokyo, 1991.

[Kim & Welch 1989] K.H. Kim and H.O. Welch, "Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications," *IEEE Trans. Comput.*, vol. 38, no. 5, pp.626-636, 1989.

[Kim & Yang 1988] K.H. Kim and S.M. Yang, "An analysis of the performance impacts of lookahead execution in the conversation scheme," in *7th Symp. Reli. Distrib. Syst.*, pp.71-81, Columbus, 1988.

[Kim & Yoon 1988] K.H. Kim and J.C. Yoon, "Approaches to implementation of a repairable distributed recovery block scheme," in *18th Int. Symp. Fault-Tolerant Comput.*, pp.50-55, Tokyo, 1988.

[Kim & You 1990] K.H. Kim and J.H. You, "A highly decentralized implementation model for the programmer-transparent coordination (PTC) scheme for cooperative recovery," in *20th Int. Symp. Fault-Tolerant Comput.*, pp.282-289, Newcastle, 1990.

[Kim 1978] K.H. Kim, "An approach to programmer-transparent coordination of recovering parallel processes and its efficient implementation rules," in *Int. Conf. Parallel Processing,* pp.58-68, 1978.

[Kim 1982] K.H. Kim, "Approaches to mechanization of the conversation scheme based on monitors," *IEEE Trans. Soft. Eng.*, vol. SE-8, no. 3, pp.189-197, 1982.

[Kim 1984] K.H. Kim, "Distributed execution of recovery blocks: an approach to uniform treatment of hardware and software faults," in *4th Int. Conf. Distrib. Comput. Syst.*, pp.526-532, 1984.

[Kim 1993] K.H. Kim, "Structuring DRB computing stations in highly decentralized systems," in *Int. Symp. Auto. Decentralized Syst.*, pp.305-314, Kawasaki, 1993.

[Kim 1995] K.H. Kim, "The distributed recovery block scheme," in *Software Fault Tolerance*, (ed. M.R. Lyu), John Wiley & Sons, pp.189-209, 1995.

[Kim et al 1976] K.H. Kim, D.L. Russell and M.J. Jenson, *Language Tools for Fault-Tolerant Programming,* Tech. Memo. PETP-1, Electronic Sciences Lab., USC, 1976.

[Kim et al 1994] K.H. Kim, L.F. Bacellar, K. Masui, K. Mori, and R. Yoshizawa, "Modular implementation model for real-time fault-tolerant LAN systems based on the DRB scheme with a configuration supervisor," *Comput. Syst. Sci. & Eng.*, vol. 9, no. 2, pp.75-82, 1994.

[Kim et al 1996] K. Kim, M.A. Vouk and D.F. McAllister, "An empirical evaluation of maximum likelihood voting in failure correlation conditions," in *ISSRE'96,* pp.330-339, 1996.

[Knight & Ammann 1991] J.C. Knight and P.E. Ammann, "Design fault tolerance," in *Software Reliability and Safety*, (eds. B. Littlewood & D. Miller), Elsevier Applied Science, pp.25-49, 1991.

[Knight & Leveson 1986a] J.C. Knight and N.G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Trans. Soft. Eng.*, vol. SE-12, no. 1, pp.96-109, 1986.

[Knight & Leveson 1986b] J.C. Knight and N.G. Leveson, "An empirical study of failure probabilities in multi-version software," in *16th Int. Symp. Fault-Tolerant Comput.*, pp.135-140, Los Alamitos, 1986.

[Knight et al 1985] J.C. Knight, N.G. Leveson and L.D.S. Jean, "A large scale experiment in *N*-version programming," in *15th Int. Symp. Fault-Tolerant Comput.*, pp.135-140, Michigan, 1985.

[Koo & Toueg 1987] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Soft. Eng.*, vol. SE-13, no. 1, pp.23-31, 1987.

[Kopetz 1974] H. Kopetz, "Software redundancy in real time systems," in *Information Processing'74, Proc. IFIP Congress*, pp.182-186, Stockholm, 1974.

[Lamport 1994] L. Lamport, "The temporal logic of actions," *ACM Trans. Prog. Lang. & Syst.*, vol. 16, no. 3, pp.872-923, 1994.

[Laprie & Kanoun 1992] J.C. Laprie and K. Kanoun,, "X-ware reliability and availability modeling," *IEEE Trans. Soft. Eng.*, vol. 18, no. 2, pp.130-147, 1992.

[Laprie 1984] J.C. Laprie, "Dependability evaluation of software systems in operation," *IEEE Trans. Soft. Eng.*, vol. SE-10, no. 6, pp.701-714, 1984.

[Laprie 1992] J.C. Laprie (ed.). *Dependability: basic concepts and terminology*, Springer-Verlag, Vienna, 1992.

[Laprie et al 1987] J.C. Laprie, J. Arlat, C. Beounes, K. Kanoun, and C. Hourtolle, "Hardware and software fault tolerance: definition and analysis of architectural solutions," in *17th Int. Symp. Fault Tolerant Comput.*, pp.116-121, Pittsburgh, 1987.

[Laprie et al 1990] J.C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Definition and analysis of hardware- and software-fault-tolerant architectures," *IEEE Comput.*, vol. 23, no. 7, pp.39-51, 1990.

[Laprie et al 1995] J.C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Architectural issues in software fault tolerance," in *Software Fault Tolerance*, (ed. M.R. Lyu), John Wiley & Sons, pp.47-80, 1995.

[Lee & Anderson 1990] P.A. Lee and T. Anderson. *Fault Tolerance: principles and practice,* Second Edition, Springer-Verlag, 1990.

[Lee 1978] P.A. Lee, "A reconsideration of the recovery block scheme," *Comput. Journal*, vol. 21, no. 4, pp.306-310, 1978.

[Lee et al 1980] P.A. Lee, N. Ghani and K. Heron, "A recovery cache for the PDP-11," *IEEE Trans. Comput.*, vol. C-29, no. 6, pp.546-549, 1980.

[Leung 1995] Y.W. Leung, "Maximum likelihood voting for fault tolerant software with finite output space," *IEEE Trans. Reliability*, vol. 44, no. 3, pp.419-427, 1995.

[Leveson 1986] N.G. Leveson, "Software safety: why, what, and how," *ACM Comput. Surveys*, vol. 18, no. 2, pp.165-170, 1986.

[Lewerentz & Lindner 1995] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: case study "Production Cell"*, LNCS-891, Springer-Verlag, Jan. 1997.

[Liggesmeyer & Rothfelder 1998] P. Liggesmeyer and M. Rothfelder, "Improving system reliability with automatic fault tree generation," in *28th Int. Symp. Fault-Tolerant Comput.*, pp.90-99, Munich, 1998.

[Lindsay et al 1984] B.G. Lindsay, L.M. Haas, C.K. Mohan, P.F. Wilms, and R.A. Yost, "Computation and communication in R*: a distributed database manager," *ACM trans. Comput. Syst.*, vol. 2, no. 1, pp.24-38, 1984.

[Liskov 1988] B. Liskov, "Distributed programming in Argus," *Communications of the ACM*, vol. 31, no. 3, pp.300-312, 1988.

[Little & Shrivastava 1998] M.C. Little and S.K. Shrivastava, "Java transactions for the internet," in *4th Conf. OO Tech. and Syst. (COOTS)*, 1992.

[Littlewood & Miller 1989] B. Littlewood and D.R. Miller, "Conceptual modeling of coincident failures in multiversion software," *IEEE Trans. Soft. Eng.*, vol. 15, no. 12, pp.1596-1614, 1989.

[Littlewood & Strigini 1993] B. Littlewood and L. Strigini, "Validation of ultra-high dependability for software-based systems," in *Communications of the ACM,* vol. 36, no. 12, 1993.

[Liu 1992] C. Liu, "A general framework for software fault tolerance," in *IEEE Workshop Fault-Tolerant Parallel & Distrib. Syst.*, Amherst, 1992.

[Lombardi 1985] F. Lombardi, "Optimal redundancy management of multiprocessor systems for supercomputing applications," in *1st. Int. Conf. Supercomput. Syst.*, pp.414-422, St. Petersbourgh, 1985.

[Lötzbeyer & Mühlfeld 1996] A. Lötzbeyer and R. Mühlfeld, "Task description of a Flexible Production Cell with real time properties," Internal FZI Technical Report, Karlsruhe, *(ftp://ftp.fzi.de/pub/PROST/projects/korsys/task_descr_flex_2_2.ps.gz)*, 1996.

[Lötzbeyer 1996] A. Lötzbeyer, "Task description of a Fault-Tolerant Production Cell," Version 1.6, available from *http://www.fzi.de/prost/projects/korsys/korsys.html*, 1996.

[Lynch et al 1993] N.A. Lynch, M. Merrit, W.E. Wehil and A, Fekete. *Atomic Transactions*, Morgan Kaufmann, 1993.

[Lyu & Avizienis 1993] M.R. Lyu and A. Avizienis, "Assuring design diversity in *N*-version software: a design paradigm for *N*-version programming," in *Dependable Computing and Fault-Tolerant Systems*, (eds. J.F. .Meyer and R.D. Schichting), Springer-verlag, Wien, Austria, pp.191-218, 1993.

[Lyu & He 1993] M.R. Lyu and Y. He "Improving the *N*-version programming process through the evolution of design paradigm," *IEEE Trans. Reliability*, vol. 42, no. 2, pp.179-189, 1993.

[Lyu 1995] M.R. Lyu (ed.). *Software Fault Tolerance,* John Wiley & Sons, 1995.

[Lyu 1996] M.R. Lyu (ed.). *Handbook of Software Reliability Engineering*, IEEE CS Press, 1996.

[Lyu et al 1995] M.R. Lyu, J. Yu, E. Keramidas, and S. Dalal, "ARMOR: analyzer for reducing module operational risk," in *25th Int. Symp. Fault-Tolerant Comput.*, pp.137-142, Pasadena, 1995.

[Madden & Rone 1984] W.A. Madden and K.Y Rone, "Design, development, integration: space shuttle primary flight software system," *Communications of the ACM*, vol.27, no. 8, pp.902-913, 1984.

[Madria 1997] S.K. Madria, "A study of the concurrency control and recovery algorithms in nested transaction environment," *Comput. Journal*, vol. 40, no. 10, pp.630-639, 1997.

[Maes 1987] P. Maes, "Concepts and experiments in computational reflection," *SIGPLAN Notices*, vol. 22, no. 12, pp.147-155, 1987.

[Mancini & Shrivastava 1989] L.V. Mancini and S.K. Shrivastava, "Replication within atomic actions and conversations: a case study in fault tolerance duality," in *19th Int. Symp. Fault-Tolerant Comput.*, pp.454-461, Chicago, 1989.

[Manna & Pnueli 1991] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, Springer, 1991.

[Marshall 1980] L.F. Marshall, *An Error Recovery Scheme for Concurrent Processes*, Ph.D. Thesis, Comput. Lab., Univ. of Newcastle upon Tyne, 1980.

[Martin 1982] D.J. Martin, "Dissimilar software in high integrity application in flight controls," in *proceedings AGARD-CP 330,* pp.36.1-36.13, 1982.

[Matos & White 1998] G. Matos and E. White, "Application of dynamic reconfiguration in the design of Fault-Tolerant Production Cell," in *4th Int. Conf. Configurable Distrib. Syst.*, Maryland, pp.2-9, 1998.

[Maxion & Olszewski 1998] R.A. Maxion and R.T. Olszewski, "Improving software robustness with dependability cases," in *28th Int. Symp. Fault-Tolerant Comput.*, pp.346-355, Munich, 1998.

[McAllister & Vouk 1996] D.F. McAllister and M.A. Vouk, "Fault-tolerant software reliability engineering," in *Handbook of Software Reliability Engineering*, (ed. M.R. Lyu), IEEE CS Press, pp.567-614, 1996.

[McAllister et al 1990] D.F. McAllister, C.E. Sun and M.A. Vouk, "Reliability of voting in fault-tolerant software systems for small output spaces," *IEEE Trans. Reliability*, vol. 39, no. 5, pp.524-534, 1990.

[Melliar-Smith & Randell 1977] P.M. Melliar-Smith and B. Randell, "Software reliability: the role of programmed exception handling," *SIGPLAN Notices*, vol. 12, no. 3, pp.95-100, 1977.

[Merlin & Randell 1978] P.M. Merlin and B. Randell, "State restoration in distributed systems," in *8th Int. Symp. Fault-Tolerant Comput.,* pp.129-134, Toulouse, 1978.

[Meyer 1992] B. Meyer. *Eiffel: the language,* Prentice Hall, 1992.

[Miller & Tripathi 1997] R. Miller and A. Tripathi, "Issues with exception handling in object-oriented systems," in *ECOOP'97*, pp.85-103, LNCS-1241, Finland, 1997.

[Miller 1989] D.R. Miller, "The role of statistical modeling and inference in software quality assurance," in *Software Certification,* (ed. B. de Neumann), Elsevier Applied Science, pp.135-152, 1989.

[Moss 1981] J.E.B. Moss. *Nested Transactions: an approach to reliable distributed computing*, Ph.D. Thesis (Tech. Report 260), MIT Lab. for Computer Science, Cambridge, 1981.

[Moulding & Barrett 1987] M.R. Moulding and P. Barrett. *An Investigation into the Application of Software Fault Tolerance to Air Traffic Control Systems: project final report,* 1049/TD.6 Version 2, RMCS, 1987.

[Nicola & Goyal 1990] V.F. Nicola and A. Goyal, "Modeling of correlated failures and community error recovery in multi-version software," *IEEE Trans. Soft. Eng.*, vol. 16, no. 3, 1990.

[Nodine & Zdonik 1984] M. Nodine and S. Zdonik, "Cooperative transaction hierarchies: a transaction model to support design applications," in *Int. Conf. VLDB'84*, pp.83-94, 1984.

[Ozaki et al 1988] B.M. Ozaki, E.B. Fernandez and E. Gudes, "Software fault tolerance in architectures with hierarchical protection levels," *IEEE Micro*, vol. 8, no. 4, pp.30-43, 1988.

[Panzl 1981] D.J. Panzl, "A method for evaluating software development techniques," *Journal Syst. Soft.*, vol. 2, 1981.

[Parrington et al 1995] G.D. Parrington, S.K. Shrivastava, S.M. Wheater and M.C. Little, "The design and implementation of Arjuna," *USENIX Comput. Syst. Journal*, vol. 8, no. 3, 1995.

[Popov & Strigini 1998] P.T. Popov and L. Strigini, "Conceptual models for the reliability of diverse systems - new results," in *28th Int. Symp. Fault-Tolerant Comput.*, pp.80-89, Munich, 1998.

[Popovic et al 1986] J.R. Popovic, D.C. Chan, D.B. Butjorjee, and B.K. Patterson, "Computer control in Candu plants," in *Int. Symp. Advanced Nuclear Service*, CAN/CNS, Toronto, 1986.

[Powell 1991] D. Powell (Ed.). *Delta-4: a generic architecture for dependable distributed computing,* Springer (Berlin), 1991.

[Preparata et al 1967] F.P. Preparata, G. Metze and R.T. Chien, "On the connection assignment problem of diagnosable systems," *IEEE Trans. Electr. Comput.*, vol. EC-16, no. 6, pp.848-854, 1967.

[Pu et al 1988] C. Pu, G.E. Kaiser and N. Hutchinson, "Split-transactions for open-ended activities," in *14th Int. Conf. VLDB*, Los Angeles, 1988.

[Pucci 1990] G. Pucci, "On the modelling and testing of recovery block structures," in *20th Int. Symp. Fault-Tolerant Comput.*, pp.353-363, Newcastle, 1990.

[Pucci 1992] G. Pucci, "A new approach to the modeling of recovery block structures," *IEEE Trans. Soft. Eng.*, vol. SE-18, no. 2, pp.356-363, 1992.

[Purtilo & Jalote 1991] J.M. Purtilo and P. Jalote, "An environment for developing fault-tolerant software," *IEEE Trans. Soft. Eng.*, vol. SE-17, no. 2, pp.153-159, 1991.

[Ramamoorthy et al 1981] C.V. Ramamoorthy, Y.R. Mok, F.B. Bastani, G. Chiu, and K. Suzuky, "Application of a methodology for the development and validation of reliable process control software," *IEEE Trans. Soft. Eng.*, vol. SE-7, no. 6, pp.537-555, 1981.

[Ramanathan & Shin 1988] P. Ramanathan and K.G. Shin, "Checkpointing and rollback recovery in a distributed system using common time base," in *7th Symp. Reli. Distrib. Syst.*, pp.13-21, Columbus, 1988.

[Randell & Xu 1993] B. Randell and J. Xu, "Object-oriented software fault tolerance: framework, reuse and design diversity," in *1st PDCS2 Open Workshop,* pp.165-184, Toulouse, 1993.

[Randell & Xu 1995] B. Randell and J. Xu, "The evolution of the recovery block concept," in *Software Fault Tolerance*, (ed. M.R. Lyu), John Wiley & Sons, pp.1-22, 1995.

[Randell 1975] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Soft. Eng.*, vol. SE-1, no. 2, pp.220-232, 1975.

[Randell 1984] B. Randell, "Fault tolerance and system structuring," in *4th Jerusalem Conf. Information Technology,* pp.182-191, Jerusalem, 1984.

[Reed 1978] D.P. Reed. *Naming and Synchronization in a Decentralized Computer System*, Ph.D. Thesis (Tech. Report 2o5), MIT Lab. for Computer Science, Cambridge, 1978.

[Romanovsky & Strigini 1995] A. Romanovsky and L. Strigini, "Backward error recovery via conversations in Ada, " *Soft. Eng. Journal*, vol. 10, no. 8, pp.219-232, 1995.

[Romanovsky et al 1996] A. Romanovsky, J. Xu and B. Randell, "Exception handling and resolution in distributed object-oriented systems," in *16th Int. Conf. Distrib. Comput. Syst.*, pp.545-552, Hong Kong, 1996.

[Romanovsky et al 1998] A. Romanovsky, J. Xu and B. Randell, "Exception handling and coordinated atomic actions in object-oriented real-time distributed systems," in *1st Int. Symp. Object-Oriented Real-Time Distrib. Comput.*, pp.32-42, Kyoto, April 1998.

[Rubira & Stroud 1994] C.M.F. Rubira and R.J. Stroud, "Forward and backward error recovery in C++," *Object-Oriented Syst.*, vol. 1, no. 1, pp.61-85, 1994.

[Russell & Tiedeman 1979] D.L. Russell and M.J. Tiedeman, "Miltiprocess recovery using conversations," in *9th Int. Symp. Fault-Tolerant Comput.*, pp.106-109, 1979.

[Russell 1980] D.L. Russell, "State restoration in systems of communicating processes," *IEEE Trans. Soft. Eng.*, vol. SE-6, no. 2, pp.183-194, 1980.

[Saglietti & Ehrenberger 1986] F. Saglietti and W. Ehrenberger, "Software diversity – some consideration about benefits and its limitations," in *IFAC SAFECOMPE'86,* pp.27-34, 1986.

[Schlichting & Thomas 1995] R.D. Schlichting and V.T. Thomas, "Programming language support for writing fault-tolerant distributed software," *IEEE Trans. Comput.*, vol. C-44, no. 2, pp.203-212, 1995.

[Schneider 1997] F.B. Schneider. *On Concurrent Programming*, Springer, 1997.

[Scott et al 1984] R.K. Scott, J.W. Gault and D.F. Mcallister, "Investigating version dependence in fault tolerant software," in *AGARD Conf. Proc. 360,* 1984.

[Scott et al 1985] R.K. Scott, J.W. Gault and D.F. Mcallister, "The consensus recovery block," in *Total Sys. Reli. Symp.*, pp.74-85, 1985.

[Scott et al 1987] R.K. Scott, J.W. Gault and D.F. Mcallister, "Fault tolerant software reliability modeling," *IEEE Trans. Soft. Eng.*, vol. SE-13, no. 5, pp.582-592, 1987.

[Shin & Lee 1984] K.G. Shin and Y. Lee, "Evaluation of error recovery blocks used for cooperating processes," *IEEE Trans. Soft. Eng.*, vol. SE-10, no. 6, pp.692-700, 1984.

[Shrivastava & Akinpelu 1978] S.K. Shrivastava and A.A. Akinpelu, "Fault-tolerant sequential programming using recovery blocks," in *8th Int. Symp. Fault-Tolerant Comput.*, pp.207, Toulouse, 1978.

[Shrivastava & Banatre 1978] S.K. Shrivastava and J.-P. Banatre, "Reliable resource allocation between unreliable processes," *IEEE Trans. Soft. Eng.*, vol. SE-4, no. 3, pp.230-241, 1978.

[Shrivastava & Wheater 1991] S.K. Shrivastava and S.M. Wheater, "Implementing fault-tolerant distributed applications using objects and multi-coloured actions," in *10th Int. Conf. Distrib. Comput. Syst.*, pp.203-210, Paris, 1991.

[Shrivastava 1978] S.K. Shrivastava, "Sequential Pascal with recovery blocks," *Software – Practice and Experience*, vol. 8, pp.177-185, 1978.

[Shrivastava 1979] S.K. Shrivastava, "Concurrent Pascal with backward error recovery: language features and examples," *Software – Practice and Experience*, vol. 9, pp.1001-1020, 1979.

[Shrivastava et al 1991] S.K. Shrivastava, G.N. Dixon and G.D. Parrington, "An overview of the Arjuna distributed programming system," *IEEE Software*, vol. 8, no. 1, pp.66-73, 1991.

[Shrivastava et al 1993] S.K. Shrivastava, L. V. Mancini and B. Randell, "The duality of fault-tolerant system structures," *Software – Practice and Experience*, vol. 23, no. 7, pp.773-798, 1993.

[Silva et al 1996] J.G. Silva, L.M. Silva, H. Madeira, and J. Bernardino, "A fault-tolerant mechanism for simple controllers," in *2nd Euro. Dependable Comput. Conf.*, pp.39-55, Taormina, 1996.

[Simon et al 1990] D. Simon, C. Hourtolle, H. Biondi, J. Bernelas, P. Duverneuil, S. Gallet, P. Vielcanet, S. DeViguerie, F. Gsell and J.N. Chelotti, "A software fault tolerance experiment for space applications," in *20th Int. Symp. Fault-Tolerant Comput.*, pp.28-35, Newcastle, 1990.

[Skarra 1989] A.H. Skarra, "Concurrency control for cooperating transactions in an object-oriented database," *ACM SIGPLAN Notices*, vol. 24, no. 4, 1989.

[Slivinski et al 1984] T. Slivinski et al. *Study of Fault-Tolerant Software Technology*. Report 2, NASA Langley Research Center, Mandex Inc., 1984.

[Spector & Gifford 1984] A. Spector and D. Gifford, "The space shuttle primary computer systems," *Communications of the ACM*, vol. 27, no. 8, 1984.

[Spector et al 1985] A.Z. Spector, J. Butcher and D. Daniels, "Support for distributed transaction in the TABS prototype," *IEEE Trans. Soft. Eng.*, vol. 11, no. 6, pp.520-530, 1985.

[Stankovic & Ramamritham 1995] J.A. Stankovic and K. Ramamritham, "A reflective architecture for real-time operating systems," in *Advances in Real-Time Systems*, (ed. S.H. Son), Prentice Hall, pp.23-38, 1995.

[Stark 1987] G.E. Stark, "Dependability evaluation of integrated hardware/software systems," *IEEE Trans. Reliability*, vol. R-36, no. 4, pp.440-444, 1987.

[Sterner 1978] B.J. Sterner, "Computerized Interlocking system – a multidimensional structure in the pursuit of safety," in *IMechE Railway Engineer Int., Nov./Dec.*, pp.29-30, 1978.

[Strigini & Avizienis 1985] L. Strigini and A. Avizienis, "Software fault-tolerance and design diversity: Past experience and future evolutions," in *IFAC SAFTCOMP'85,* pp.167-172, 1985.

[Strigini 1990] L. Strigini, "Software Fault-Tolerance," *ESPRIT Project 3092 PDCS1, 1st Year Report,* vol. 2, pp.1-39, Toulouse, 1990.

[Strigini et al 1997] L. Strigini, F. Di Giandomenico and A. Romanovsky. "Coordinated backward recovery between client processes and data servers," *IEE Proc. – Soft. Eng.*, vol. 144, no. 2, pp.134-146, 1997.

[Stroud & Wu 1995] R.J. Stroud and Z. Wu, "Using metaobject protocols to implement atomic data types," in *Proc. ECOOP'95*, LCNS 952, pp. 165-189, 1995.

[Stroustrup 1991] Stroustrup. *The C++ Programming Language,* Addison Wesley, 1991.

[Subramanian & Tsai 1996] S. Subramanian and W. Tsai, "Backup pattern: designing redundancy in object-oriented software," in *Pattern Languages of Program Design,* (eds. J. Coplien & D.C. Schnidt), Addision-Wesley, 1996.

[Sullivan & Chillarege 1991] M.S. Sullivan and R. Chillarege, "Software defects and their impact on system availability – a study of field failures in operating systems," in *21st Int. Symp. Fault-Tolerant Comput.*, pp.2-9, Montreal, 1991.

[Sullivan & Masson 1990] G.F. Sullivan and G.M. Masson, "Using certification trails to achieve software fault tolerance," in *20th Int. Symp. Fault-Tolerant Comput.*, pp. 423-431, Newcastle, 1990.

[Sullivan & Masson 1991] G.F. Sullivan and G.M. Masson, "Certification trails for data structures," in *21st Int. Symp. Fault-Tolerant Comput.*, pp. 240-247, Montreal, 1991.

[Sweet 1995] W. Sweet, "The glass cockpit," *IEEE Spectrum,* pp.30-38, 1995.

[Tai 1994] A. Tai, "Performability-driven adaptive fault tolerance," in *24th Int. Symp. Fault-Tolerant Comput.*, pp.496-503, 1994.

[Tai et al 1993] A. Tai, A. Avizienis and J. Meyer, "Evaluation of fault-tolerant software: a performability modeling approach," in *Dependable Computing for Critical Applications 3,* (eds. C.E. Landweh, B. Randell & L. Simoncini), pp.113-135, Sprinter-Verlag, 1993.

[Taylor & Black 1985] D.J. Taylor and J.P. Black, "Guidelines for storage structure error correction," in *15th Int. Symp. Fault-Tolerant Comput.*, pp.20-22, Michigan, 1985.

[Taylor 1986] D.J. Taylor, "Concurrency and forward recovery in atomic actions," *IEEE Trans. Soft. Eng.*, vol. 12, no. 1, pp.69-78, 1986.

[Taylor et al 1980] D.J. Taylor, D.E. Morgan and J.P. Black "Redundancy in data structures: improving software fault tolerance," *IEEE Trans. Soft. Eng.*, vol. SE-6, no. 6, pp.585-593, 1980.

[Taylor et al 1994] P. Taylor, V. Cahill and M. Mock, "Combining object-oriented systems and open transaction processing," *Comput. Journal*, vol. 37, no. 6, pp.487-498, 1994.

[Theuretzbacher 1986] N. Theuretzbacher, "VOTRICS: voting triple-Modular computing system," in *16th Int. Symp. Fault-Tolerant Comput.*, pp.144-150, Vienna, 1986.

[Thomasian 1998] A. Thomasian, "Concurrency control: methods, performance, and analysis," *ACM Comput. Surveys*, vol. 30, no. 1, pp.70-119, 1998.

[Tomek et al 1993] L.A. Tomek, J.K. Muppala and K.S. Trivedi, "Modeling correlation in software recovery blocks," *IEEE Trans. Soft. Eng.*, vol. 19, no. 11, pp.1071-1086, 1993.

[TransArc 1997] TransArc Corporation, *Encina*.
(*http://www.transarc.com/afs/transarc.com/public/www/public/www/Public/prodServ/Product/Encina*)

[Traverse 1987] P. Traverse, "AIRBUS and ATR system architecture and specification," in *Dependable Computing and Fault-Tolerant Systems*, (eds. A. Avizienis, H. Kopetz & J.C. Lapire), Springer-Verlag, Wien, New York, pp.95-104, 1987.

[Troy & Baluteau 1985] R. Troy and C. Baluteau, "Assessment of software quality for the airbus A310 automatic pilot," in *15th Int. Symp. Fault-Tolerant Comput.*, pp.438-443, 1985.

[Tso & Avizienis 1987] K.S. Tso and A. Avizienis, "Community error recovery in *N*-version software: a design study with experimentation," in *17th Int. Symp. Fault-Tolerant Comput.*, pp.127-133, 1987.

[Tso & Shokri 1996] K.S. Tso and E.H. Shokri, "An integrated environment for development and testing of software fault tolerance systems," in *Int. Workshop CAD, Test & Evaluation for Dependability (CADTED'96)*, pp.66-71, Beijing, 1996.

[Turner et al 1987] D.B. Turner, R.D. Burns and H. Hecht, "Designing micro-based systems for fail-safe travel," *IEEE Spectrum,* vol. 24, no.29, pp.58-63, 1987.

[Tyrrell & Carpenter 1992] A.M. Tyrrell and G.F. Carpenter, "The specification and design of atomic actions for fault tolerant concurrent software," *Microprocessing & Microprogramming*, vol. 35, pp.363-368, 1992.

[Tyrrell & Carpenter 1995] A.M. Tyrrell and G.F. Carpenter, "CSP methods for identifying atomic actions in the design of fault-tolerant concurrent systems," *IEEE Trans. Soft. Eng.*, vol. SE-21, no. 7, pp.629-639, 1995.

[Tyrrell & Holding 1986] A.M. Tyrrell and D.J. Holding, "Design of reliable software in distributed systems using the conversation scheme," *IEEE Trans. Soft. Eng.*, vol. SE-12, no. 9, pp.921-928, 1986.

[Vachon et al 1998] J. Vachon, D. Buchs, M Buffo, G.D.M. Serugendo, B. Randell, A. Romanovsky, R.J. Stroud, and J. Xu, "COALA – A formal language for coordinated atomic actions," 3rd Year Report, *ESPRIT Project 20072 on Design for Validation*, pp.43-86, 1998.

[Voges 1987] U.Voges (ed.). *Dependable Computing and Fault-Tolerant Systems*, (eds. A. Avizienis, H. Kopetz & J.C. Lapire), Springer-verlag, Wien, New York, 1987.

[Vouk et al 1985] M.A. Vouk, D.F. McAllister and K.C Tai, "Identification of correlated failure of fault-tolerant software systems," in *Proceedings COMPSAC'85*, pp.437-444 , 1985.

[Vouk et al 1993] M.A. Vouk, D.F. McAllister, D.E. Eckhardt and K. Kim, "An empirical evaluation of consensus voting and consensus recovery block reliability in the presence of failure correlation," *Journal Comput. & Soft. Eng.*, vol. 1, no. 4, pp.367-388, 1993.

[Warne 1994] J. Warne, "Flexible transaction framework for dependable workflows," *ANSA Consortium*, 1994.

[Weihl 1989] W.E. Weihl, "Local atomicity properties: modular concurrency control for abstract data types," *ACM Trans. Prog. Lang. Syst.*, vol. 11, no. 2, pp.249-282, 1989.

[Weikum & Schek 1992] G. Weikum and H.J. Schek, "Concepts and applications of multilevel transactions and open-nested transactions," in *Transaction Models for Advanced Database Applications* (ed. A. Elmagarmid), Morgan-Kaufman, pp.516-546, Feb. 1992.

[Welch 1983] H.O. Welch, "Distributed recovery block performance in a real-time control loop," in *Real-Time Syst. Symp.*, pp.268-276, Virginia, 1983.

[Wellings & Burns 1997] A.J. Wellings and A. Burns, "Implementing Atomic Actions in Ada 95", *IEEE Trans. Soft. Eng.*, vol. 23, no. 2, pp.107-123, 1997.

[Williams et al 1983] J.F. Williams, L. J. Yount and J.B. Flanningan, "Advanced autopilot flight director system computer architecture for Boeing 737-300 aircraft," in *5th Digital Avionics Conf.*, Seattle, 1983.

[Wing et al 1992] J.M. Wing, M. Faehndrich, J.G. Morrisett, and S.M. Nettles, "Extensions to standard ML to support transactions," in *ACM SIGPLAN Workshop ML & Appli.*, pp.1-15, June 1992.

[Wing 1993] J.M. Wing, "Decomposing and recomposing transactional concepts," in *Object-Based Distributed Programming – ECOOP'93 Workshop* (eds. R. Guerraoui, O. Nierstrasz & M. Riveill), Springer-Verlag, pp.111-121, 1993.

[Wood 1981] W. Wood, "A decentralised recovery control protocol," in *11th Int. Symp. Fault-Tolerant Comput.*, pp.159-164, 1981.

[Wright 1986] N.C.J. Wright, "Dissimilar software," in *Workshop Design Diversity in Action*, (see [Voges 1987]), Baden, 1986.

[Wu & Fernandez 1994] J. Wu and E.B. Fernandez, "Using Petri nets for the design of conversation boundaries in fault-tolerant software," *IEEE Trans. Parallel Distrib. syst.*, vol. 5, no. 10, pp.1106-1112, 1994.

[Xu & Huang 1990] J. Xu and S. Huang, "A new comparison-based scheme for multiprocessor fault tolerance," *Microprocessing & Microprogramming*, vol. 30, no. 1-5, pp.617-623, 1990.

[Xu & Huang 1995] J. Xu and S. Huang, "Sequentially t-diagnosable systems: a characterization and its applications," *IEEE Trans. Comput.*, vol. 44, no. 2, *Special Issue on Fault-Tolerant Computing*, pp.340-346, 1995.

[Xu & Randell 1997] J. Xu and B. Randell, "Software fault tolerance: $t/(n-1)$-variant programming," *IEEE Trans. Reliability*, vol. 46, no. 1, pp.60-67, 1997.

[Xu 1991] J. Xu, "The $t/(n-1)$-diagnosability and its application to fault tolerance," in *21st Int. Symp. Fault Tolerant Comput.*, pp.496-503, Montreal, 1991.

[Xu et al 1995a] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu, "Fault tolerance in concurrent object-oriented software through coordinated error recovery," in *25th Int. Symp. Fault-Tolerant Comput.*, pp.499-508, Pasadena, 1995.

[Xu et al 1995b] J. Xu, A. Bondavalli and F. Di Giondomenico, "Dynamic adjustment of dependability and efficiency in fault-tolerant software," in *Predictably Dependable Computing Systems* (eds. B. Randell, J.C. Laprie, H. Kopetz & B. Littlewood), Springer-Verlag, pp.155-172, 1995.

[Xu et al 1995c] J. Xu, B. Randell, C.M.F. Rubira-Casavara and R.J. Stroud, "Toward an object-oriented approach to software fault tolerance," in *Recent Advances in Fault-Tolerant Parallel and Distributed Systems* (eds. D.K. Pradhan and D.R. Avresky), IEEE CS Press, pp.226-233, 1995.

[Xu et al 1996] J. Xu, B. Randell and A.F. Zorzo, "Implementing software-fault tolerance in C++ and Open C++," in *Int. Workshop CAD, Test & Evaluation for Dependability (CADTED'96)*, pp.224-229, Beijing, 1996.

[Xu et al 1998a] J. Xu, A. Romanovsky, and B. Randell, "Coordinated exception handling in distributed object systems: from model to system implementation," in *18th Int. Conf. Distrib. Comput. Syst.*, pp.26-29, Amsterdam, 1998.

[Xu et al 1998b] J. Xu, A. Romanovsky, A. Zorzo, B. Randell, R.J. Stroud and E. Canver, "Developing control software for Production Cell II: failure analysis and system design using CA actions," 3rd Year Report, *ESPRIT Project 20072 on Design for Validation*, pp.167-188, 1998.

[Yang & Kim 1992] S.M. Yang and K.H. Kim, "Implementation of the conversation scheme in message-based distributed computer systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 5, pp.555-572, 1992.

[Yonezawa & Watanabe 1989] A. Yonezawa and T. Watanabe, "An introduction to object-based reflective concurrent computation," *SIGPLAN Notices*, vol. 24, no. 4, pp.50-53, 1989.

[Yount 1986] L.J. Yount, "Use of diversity in Boeing airplanes," in *Workshop Design Diversity in Action*, (see [Voges 1987]), Baden, 1986.

[Zorzo et al 1999] A.F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R.J. Stroud and I.S. Welch, "Using coordinated atomic actions to design safety-critical systems: a production cell case study," *Software – Practice & Experience*, vol. 29, no. 28, pp.677-697, 1999.