COMPUTING SCIENCE

Object-Oriented Construction of Fault-Tolerant Software

J. Xu and B. Randell

TECHNICAL REPORT SERIES

No. 444 July, 1993
Object-Oriented Construction of Fault-Tolerant Software

J. Xu and B. Randell

Abstract
Software fault tolerance is often necessary, but itself can be dangerously error-prone because of the additional effort that must be involved in the programming process. Adding redundancy to programs may increase the size and complexity and thus adversely affect software reliability. Object-oriented programming provides a particularly appropriate framework, based on the theory of abstract data types, for enforcing reliability and controlling complexity. This paper introduces an approach to achieving software fault tolerance in object-oriented systems in a disciplined manner and with reduced cost, and discusses the problem of how to incorporate fault tolerance into concurrent object-oriented computing.
Bibliographical details

XU, Jie

Object-Oriented Construction of Fault-Tolerant Software
[By] J. Xu and B. Randell


(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no.444)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
RANDELL, Brian

Abstract

Software fault tolerance is often necessary, but itself can be dangerously error-prone because of the additional effort that must be involved in the programming process. Adding redundancy to programs may increase the size and complexity and thus adversely affect software reliability. Object-oriented programming provides a particularly appropriate framework, based on the theory of abstract data types, for enforcing reliability and controlling complexity. This paper introduces an approach to achieving software fault tolerance in object-oriented systems in a disciplined manner and with reduced cost, and discusses the problem of how to incorporate fault tolerance into concurrent object-oriented computing.

About the author

Mr. Jie Xu joined the Department of Computing Science in April 1990 where he is a Research Associate.

Professor Randell has been a Professor of Computing Science in the Department of Computing Science since 1969.

Suggested keywords

ABstraction
CONCURRENCY
INHERITANCE
OBJECT-ORIENTED PROGRAMMING
PROGRAM STRUCTURING
SOFTWARE FAULT TOLERANCE
SOFTWARE RELIABILITY

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.6425
U.D.C. 681.322.06
Object-Oriented Construction of
Fault-Tolerant Software

Jie Xu and Brian Randell

Department of Computing Science
University of Newcastle upon Tyne

ABSTRACT
Software fault tolerance is often necessary, but itself can be dangerously error-
prone because of the additional effort that must be involved in the programming
process. Adding redundancy to programs may increase the size and complexity
and thus adversely affect software reliability. Object-oriented programming
provides a particularly appropriate framework, based on the theory of abstract
data types, for enforcing reliability and controlling complexity. This paper
introduces an approach to achieving software fault tolerance in object-oriented
systems in a disciplined manner and with reduced cost, and discusses the
problem of how to incorporate fault tolerance into concurrent object-oriented
computing.

Key words--Abstraction, concurrency, inheritance, object-oriented programming, program
structuring, software fault tolerance, software reliability.

This work has been supported by funding from the ESPRIT Basic Research Action 6362 entitled "Predictably
Dependable Computing Systems (PDCS2)".
1. Introduction

Software dependability now constitutes a major problem since software errors have become a common cause of unreliability in many computing systems [And85]. Software design fault tolerance is often necessary, but can be error-prone: redundancy of design (not simple replication of programs) and extra effort are required in programming process. Adding redundant code to programs could increase the software's complexity and thus lead to a decrease, rather than an increase, in reliability. In order to improve the effectiveness and acceptance of software fault tolerance, redundancy within a system, especially a concurrent system, must be carefully structured and controlled to minimise any increase in complexity. The purpose of this paper is to explore the application of some new ideas in program structuring, and particularly in object-oriented programming [Mey88], to the provision of software fault tolerance.

Object-oriented programming techniques, if used well, reduce software complexity and improve dependability and maintainability of complex systems by decomposing them into manageable pieces with well-defined functionality. The various forms of software fault tolerance that have been devised to date either require special new programming language features (provided by a special compiler or pre-processor) or require strict, but unchecked, adherence by a programmer to a set of special programming conventions. The former approach cuts one off from the mainstream of programming language developments, the latter can be dangerously error-prone. Our aim is to show how these forms of software fault tolerance can in fact be provided in general purpose languages and within a standard framework, and that the use of some of the newer techniques in object-oriented programming can greatly facilitate and simplify the task of providing and deploying such software fault tolerance by promoting reusability, and can avoid requiring programmers to adhere to special sets of unchecked conventions.

The sets of components arising from object-oriented programming practices, if chosen carefully, may be used in the construction of several related applications. The achievement of such software reuse can achieve significant benefits, relating both to software quality and software costs. However, for reusable components, the potential consequences of incorrect behavior are even more serious than for application-specific developments. What is needed is a more systematic approach which helps to carefully structure redundancy and achieve reuse in order to reduce the opportunities of software faults.

The paper is organized as follows. Section 2 gives an overview of software fault tolerance concepts. Section 3 introduces our abstraction model which provides an object-oriented linguistic framework for describing various forms of software fault tolerance in a well-coordinated and disciplined fashion. Our approach is not dependent on any specific object-oriented language; here a C++-like notation [Stro86] is simply used as a means to explain our object software construction. Section 4 discusses the delicate problem of how to attain fault tolerance in concurrent object-oriented systems. The final section presents conclusions.
2. Software Fault Tolerance Concepts

Software is a crucial component of all computing systems and often the major cause of unreliability. Software fault tolerance is concerned with techniques necessary to enable a system to tolerate software 'bugs' or faults, that is, faults in the design and construction of the software itself. Appropriate techniques exist and have been proved successful. In what follows, we shall only take some canonical proposals and methods as examples to explain software fault tolerance concepts. Strigini presented in [Str90] a comprehensive survey of software fault tolerance issues, where further references can be found.

2.1. Exception Handling, Atomicity and Error Recovery

Using exception-handling mechanisms to deal with abnormal cases of system behavior is a classical programming idea. Exceptions and exception handling can provide a suitable framework for structuring the fault tolerance activities incorporated in a system, allowing a clear separation of the abnormal activities of the system from its normal activity so as to avoid the pollution of program structure implied by the mixture of normal processing and handling of abnormal cases. During program execution, various abnormal events may occur: for example, an improperly written software component may produce an unacceptable outcome. The detection of such events will usually trigger a signal, or exception, which interrupts the normal flow of execution. If the system's text does not include any handler for the exception, program execution will terminate. Exception handlers may, however, be provided in order to handle exceptions, if possible, resuming execution after correcting the cause of the exception.

A system can be viewed here as a set of components interacting under the control of a design (which is itself a component of the system) [LA90]. Clearly, the system model is recursive in that each component can itself be considered as a system in its own right and thus may have an internal design which can identify further subcomponents. Components receive requests for service and produce responses. When a component cannot satisfy a request for service, it will return an exception. An idealised fault-tolerant component should include both normal and abnormal responses in the interface between interacting components. It can either deal with exceptional responses raised by components at a lower level or else propagate the exception to a higher level of the system (see Fig.1.3 in [And85]).

There may be a substantial delay between the erroneous state transition caused by a fault and the detection of any error, during which damage can spread as a result of any subsequent flow of information. A useful concept for structuring the activity of a system and for confining the flow of erroneous information is that of an atomic action. An atomic action is the activity of a group of system components such that there are no interactions between that group and the rest of the system for the duration of the activity. By imposing constraints on component interactions, thereby imposing structure on the flow of information (particularly, erroneous information concerned with fault tolerance) within the system, atomic actions become particularly helpful in establishing precisely the extent to which the system state has been damaged before appropriate error recovery can be undertaken. The recovery itself can be categorized as follows.

Forward error recovery aims to identify the error and, based on this knowledge, correct the system state containing the error. Such an approach demands some understanding of the errors
which exist. In contrast, *backward error recovery* aims to correct the system state by restoring
the system to a state which occurred prior to the manifestation of the fault. No knowledge of
the errors in the system state is required. Since software faults are by their nature
unpredictable, as are the errors which they introduce, backward error recovery is the most
generally applicable for software fault tolerance. However, for those limited cases where the
characteristics of a fault are well understood, forward recovery can provide a more efficient
solution.

The objective of software fault tolerance is to prevent software faults from causing system
failure. All fault tolerance techniques depend on redundancy being added to the system. This
redundancy can be regarded as performing four major activities: error detection, damage
assessment (achieved by having damage-confinement structures, for example, atomic actions),
error recovery and fault treatment (i.e., fault removal) [LA90]. Needless to say, the
redundancy required to be able to tolerate software faults is not simple replication of programs
but redundancy of design. The various approaches to software fault tolerance can be in general
divided into two categories: masking redundancy and dynamic redundancy.

### 2.2. Masking Redundancy

Masking redundancy, also known as “static redundancy”, uses extra software components
(called versions or variants) within a system such that the effects of one or more software
errors are masked from, and not perceived by, the environment of that system. The standard
method employed to obtain software fault masking is *n*-version programming [AC77], a newer
one being *t*/(*n*-1)-variant programming [Xu91, XR92].

The *n*-version programming scheme can be regarded as a direct extension of NMR structures
used in hardware. *N* versions of a program which have been independently designed to satisfy
a common specification are executed (in parallel) and their results compared by some form of
replication check. Based on a majority vote, this check can pass on the (presumed to be correct)
results generated by the majority to the rest of the system, and therefore mask faults (i.e. the
results produced by the minority). The *t*/(*n*-1)-VP approach is also based on the parallel
execution of *n* versions (or variants), but exploits a particular diagnosability measure, *t*/(*n*-1)-
diagnosability, developed in the area of system-level fault diagnosis for selecting presumably
correct results as the output. This technique isolates the faulty software components that might
have produced incorrect results within a set of at most (*n*-1) variants, so that software faults in
the set can be effectively masked under certain conditions even if incorrect results form the
majority of *n* results.

### 2.3. Dynamic Redundancy

A system with dynamic redundancy consists of several redundant components with just one
active at a time. If a software error is detected in the active component it is replaced by a spare
component. Dynamic redundant systems can be further divided into two categories: cold-
standby systems and hot-standby systems. Three examples of the use of dynamic redundancy
are recovery blocks [Ran75], *n*-self checking programming [LAB+87], and self-configuring
optimal programming [BDX92; XBD93].
In the recovery block approach, normally only the first variant (called the primary alternate) is operating and an acceptance test is applied to its result for the purpose of error detection: if the result fails to pass the test, the state of the system is restored and the second variant is invoked on the same input data, and so on sequentially until either the result from a variant passes the acceptance test or all the variants are exhausted. N-self-checking programming provides fault tolerance through the parallel execution of $n$ self-checking software components. Each self-checking component is constructed either from a single variant and an acceptance test or from the association of a pair of variants with a comparator. One component is regarded as the active component, and the others are considered as "hot"-standby spares. Upon failure of the active component, service delivery is switched to a "hot" spare. The self-configuring optimal programming scheme configures and executes variants in a dynamic and adaptive fashion in order to make the best use of available resources. It organizes the execution of variants in phases, dynamically constructing a currently active set $V_i$ (composed of $a$ or more variants) at the beginning of the $i$th phase. An adjudication is made at the end of every phase. Once conditions for the release of a result are met, the result will be output immediately and further phases avoided.

2.4. Concurrent System Recovery

It is important that techniques of error recovery be extended to concurrent systems because in practice many programs, such as operating systems and real-time control systems, are concurrent. Such a system typically consists of a set of communicating sequential processes which execute in parallel and cooperate to achieve some goal. These systems are particularly prone to error since they are usually extremely complex. The incorporation of fault tolerance may be a practical method of improving their dependability [GK89].

Fault tolerance in concurrent systems cannot be attained merely by using some fault tolerance scheme in each separate process. For example, in a system of interacting processes, the recovery of only one process to its checkpoint can lead to an inconsistent global system state, unless all other relevant processes are rolled back as well. But, if the recovery points for all of the processes involved are not carefully coordinated, the entire software system could find itself rolled back to its initial state -- this is the domino effect described in [Ran75]. A well-known method for avoiding the domino effect is that of conversations [Ran75, Kim82].

In a conversation, which can be regarded as a recoverable atomic action, the set of participating processes may communicate freely between each other but with no other outside processes. Processes may enter the conversation at different times but, on entry, should establish checkpoints. If a process within a conversation fails all the participating processes are rolled back to the respective checkpoints at the start of the conversation and each executes an alternate algorithm. Clearly, the 'distance' of the rollback is limited and thus no domino effect could result. A survey of various methods of dealing with the domino effect and a summary of the related problems of program structure can be found in [GK89].

2.5. Problems and Difficulties

Each of the schemes listed above and many other schemes may be suited for a class of applications and may have features which complement deficiencies in some of the others. Their
potential for enhancing software reliability has been demonstrated by various experiments. These techniques, however, often defeat their own purposes:

(1) Fault tolerance involves adding redundancy to the system. An increase in complexity may cause the design of fault-tolerant software itself to be an error-prone process. Complexity control is crucial here, especially in concurrent systems.

(2) Mainstream programming languages do not at present address software fault tolerance. The special language features required for the provision of software fault tolerance or set of complex programming conventions that must be adhered by the application programmer could limit application areas and be liable to introduce new software faults.

(3) The provision of software fault tolerance capability is done mostly in an ad hoc manner, rather than in a disciplined manner. There is no a unified, standard language structure for various forms of software fault tolerance that could lead to systematic and safe use of the existing schemes.

(4) High development costs pose a real question, relative to approaches based on striving to preclude the presence of faults in design.

To solve some of these problems, [Liu91] proposed a design notation for a wide class of fault-tolerant software structures and [BS91] introduced a description language, based on a data-flow model, suitable for describing fault-tolerant schemes in application software. Their methods, however, require a new programming language or special language features provided by a preprocessor. We believe that object-oriented programming is a particularly appropriate approach to the problem of enforcing dependability, controlling complexity, building abstraction and reducing cost. Unfortunately, few researches have explored the potential benefits of using object-oriented techniques to facilitate software fault tolerance and fewer researches have attempted to incorporate fault tolerance into concurrent object-oriented systems. We will demonstrate how our approach attacks all the problems listed above.

3. An Object Framework for Software Fault Tolerance

This section first describes a simple abstract framework for expressing software fault tolerance schemes in programs and then explains the object-oriented construction of the framework. The approach we take could in principle be used with any language which provides higher order functions, but casting it in object-oriented terms allows us to make our chosen classification hierarchy explicit.

3.1. An Abstract Framework

In order to facilitate and simplify the production of fault-tolerant software, we must first establish an abstract framework of a fault-tolerant software system in which various forms of fault-tolerant structures can be combined in coordination and expressed systematically. Our framework has the following characteristics:
(1) It is based on the recursive system model and the concept of idealised fault-tolerant components so that the complexity of a fault-tolerant software system can be well controlled.

(2) The implementation details of various schemes or structures are made transparent to the application programmer in the hope that the extra burden of fault tolerance design can be limited and thus the possibility of introducing faults into the software reduced.

(3) The framework provides the support information and offers the programmer a choice of the execution mode of software variants (e.g., between sequential, parallel and dynamic execution of variants) and, therefore, requirements for dependability, timeliness and efficiency can be considered comprehensively at the construction time.

(4) It further presents the programmer with explicit and flexible interfaces for the selection and combination of different fault-tolerant structures.

(5) The framework is itself simple, we would claim.

To be more exact, we propose here an abstraction model of a fault-tolerant software component which is composed of three parts: redundant variants of diverse design, an adjudicator [And86] and a control architecture. Figure 1 gives an illustration of the framework, where variants deliver the same service through independent designs and implementations, the adjudicator selects a single, presumably correct result from the set of results produced by variants, and the control architecture is in charge of the invocation of variants, controls their execution and determines the overall system output with the aid of the adjudicator.

![Diagram](image)

**Figure 1. Abstract framework for software fault tolerance**

The adjudicator in the framework is the main method of achieving error detection. However, in practice it will not detect all errors and should be supplemented with assertion statements within a software design and with detection mechanisms associated with the hardware (e.g., checking invalid memory access by the software). Moreover, the adjudication component is usually
considered as the hardcore of a fault-tolerant structure and, consequently, should be kept as simple as possible in order to achieve high reliability. This is why we separate in this framework the control function, which can be quite complex [DS90], from the adjudicator.

In principle each variant in the framework has a set of exception handlers associated with it, and can be regarded as an idealised fault-tolerant component. In such components, three classes of exceptional situation are distinguished: an interface exception is signalled when interface checks determine that an invalid service request has been made to the variant and the part of the system that made the invalid request must deal with the exception; a local exception is signalled when the variant has detected an error that its own exception handlers should deal with; and a failure exception is the means by which the variant notifies the control architecture that it has been unable to provide the service requested of it.

However exception handling can also be provided for the adjudicator. Thus, it too can be viewed as an idealised fault-tolerant component. The architecture itself can also have a set of exception handlers associated with it - or more exactly with the combination of architecture plus adjudicator plus variants. The whole structure is therefore in principle fully recursive.

An ideal fault-tolerant component, in general a software component, can conveniently be thought of as an object [LA90]. Similarly to such components, objects have a well-defined external interface that provides operations to manipulate an encapsulated internal state. However, the object-oriented approach emphasises the use of classes and inheritance. An object is an instance of some class or type. Therefore, based on the concept of abstract data type, it is natural to describe those components in our framework in terms of three distinct classes, respectively corresponding to variants, adjudicators and control architectures, and this is the approach we take in this paper. We regard the classes described in this paper as low-level tools. Higher-level classes for the implementation of a more sophisticated scheme could readily be developed using inheritance and redefinition. Of course, there are many other ways of implementing this framework in an object-oriented fashion. However, our classes are intended to be built on top of a language such as C++ or Eiffel [Mey92] without modification of the compiler, so that rapid and instructive experiments are possible.

3.2. Object-Oriented Construction of the Framework

We use here our C++-like notation to express the object-oriented construction of the framework in order to demonstrate how software fault tolerance can in fact be provided in general purpose (or so-called mainstream) languages, though our classes can also be simply described in other languages.

Before introducing our object-oriented construction, it is necessary to first explain the notion of data abstraction, type inheritance and polymorphism. Both data abstraction and type inheritance are based on the class concept. A class is a program construct describing the behaviour of a family of objects. The type inheritance mechanism of C++ works as follows: given a base class, a derived class of the base class can be defined so that it inherits some or all of the attributes of the base class. Polymorphism refers to the situation in which objects belonging to different classes can respond to the same message (or operation), usually in different ways.
3.2.1. Software Variants

We first introduce the variant class:

```cpp
class variant
{
    .... ....  //private variables and operations
public:
    virtual void variantDefinition();  //interface
    virtual void exceptionHandler1(...);
    virtual void exceptionHandler2(...);
    .... ....
};
```

The function of the variant class is to provide an interface through which the application programmer can develop particular concrete program variants (e.g., variants of the sorting operation) and, furthermore, the control architecture can organise the execution of these user-defined variants so as to provide either static or dynamic redundancy, as chosen by the application designer. Class variant can be viewed here as an abstract base class: it is used only for deriving other classes, and not for creating class objects. A dummy definition is provided for the variantDefinition() function that is intended to be overridden in the derived classes. (For virtual functions, it is possible to declare a function without providing any definition.) Therefore, polymorphism is implemented via virtual functions: the application programmer defines these virtual functions, according to application requirements, as concrete ones in the derived classes of the abstract base class. The abstract base class may provide a set of standard exception handlers (e.g., address out of range, divide by zero, invalid operation code) which deal with some local errors detected during the execution of the user-defined variants.

The application programmer can then write a concrete variant, say, quickSort, in the form of a derived class:

```cpp
class quickSort : public variant
{
    .... ....  //private attributes and class invariant
public:
    int variantDefinition(int);  //user-defined sorting
    .... ....  //and other operations
    void exceptionHandler1(...); //user-defined handlers
    void exceptionHandler2(...);
    .... ....
};
```

Clearly, there is no essential difference between the special class as a variant in the framework and a general-purpose class used in non-fault-tolerant applications. This is just what we expect: no special programming conventions need to be adhered to. For the purpose of complexity control, the exception handlers, which deal with local and interface exceptions during the execution of operations, are separated from the body of operations (i.e., normal processing) and are declared in the declaration of the class. Further, the Eiffel contract model [Mey92], or the Eiffel assertion mechanisms, may be incorporated into the construction so as to enforce reliability in a disciplined manner. A precondition and a postcondition may be associated with each operation and a class invariant may be used to constrain all the operations of a given class. The original C++ does not provide the syntax support for these mechanisms, but we do not
consider this to be a fundamental problem. For example, the same mechanisms can be implemented by defining particular classes and operations.

It is worth noting that a programmer who wants to call a sort operation does not have to write \texttt{variantDefinition} instead of a more natural name like \texttt{sort}. A simple way of arranging this is to use the name \texttt{sort} to indicate the operation and write the \texttt{variantDefinition} operation as a user (or a caller) of the \texttt{sort} operation.

The application programmer may develop a set of specific variants in the form of a hierarchy. However, while sharing some common functionality, diversity among the variants must be carefully taken into account. If the state or operations intended to be shared may lead a big danger to reliability in the form of common faults, alternate implementations are required. What we need is a systematic method for constructing such a hierarchy. The hierarchy may be formed into a tree with the \texttt{variant} class as its root. Without sharing any common functionality and permitting any inheritance relationship, a set of independently-designed variants is required to be placed at the different branch of or along the horizontal direction of the hierarchy; and the improved or modified form of a variant, which might have shared some attributes of the variant, can be placed at a level below it but at the same branch. Following this method, inheritance and polymorphism, two important but potentially unsafe aspects of object-oriented programming, can be correctly exploited in a disciplined manner, ensuring that variants which share some common functionality with each other are not used in the same fault-tolerant structures so that common software faults can be avoided.

In fact, the programmer does not necessarily produce the code of variants by him or herself: the object-oriented programming paradigm suggests the possibility of reusing the code in other applications. Suppose that a variant needs an operation sorting an array, and such an operation exists in a component library of other applications. It might be possible for the programmer to construct the variant simply by inheriting the operation directly, without rewriting the code. The multiple inheritance mechanism provides convenient support for this purpose. A user-defined class, say, \texttt{variant 2} (as shown later in Figure 2), can be a subclass of both the abstract base class, used as an interface to the control architecture, and other classes such as the classes defined in other applications. It can therefore inherit all the state and operations of its parent classes. Of course, in practice we are seldom able to reuse a software component exactly as it stands: most of the time, some local adjustments are needed. Inheritance with redefinition provides a particularly appropriate degree of flexibility in this aspect.

3.2.2. Adjudication algorithms

A basic set of adjudication algorithms (e.g., majority voting based on equality checks, and simple reasonableness checks) can be provided by the system. In practice, the adjudication algorithms are usually application-specific; they are thus a user-provided error detection measure and can be as reliable or as complicated as the user wishes. Following our approach to the development of software variants, the application programmer can easily insert such a measure to a program. We now provide the \texttt{adjudicator} class as follows, again as an abstract base class.
class adjudicator
{ ... ... //private variables and operations
public:
    virtual boolean acceptanceTest(); //standard adjudication
    virtual void voting(...); //operations
    ... ...
    void exceptionHandler1(...); //handlers for errors in
    void exceptionHandler2(...); //adjudicators
    ...
};

Although it is hoped that the adjudicator can be kept simple, abnormal events might still occur during its execution. Exception handling measures can therefore be provided as part of the adjudicator. Moreover, it is possible for the adjudicator to receive an exception from a variant rather than a valid result. In this case, it may deal with the exception or simply propagate it to a component at the higher level of the system (e.g., the control architecture or program).

3.2.3. The Control Architecture

The function of the control architecture is to control the execution of variants, to invoke the adjudication operation and to output the desired results, if any exist, or to report an exception to the next higher level of the system. The various standard control architectures would be provided by the system.

enum sftStatus {NORMAL, EXCEPTION, FAILURE...}

class sftFramework
{ ... ... //private variables and operations
    adjudicator* pa; //pointer to adjudicator
    variant* pv1; //pointers to variants
    variant* pv2;
    ... ...
    public:
    sftStatus recoveryBlock(...); //execution modes
    sftStatus nVersionProgramming(...);
    sftStatus sequentialExecution(...);
    sftStatus dynamicExecution(...);
    ... ...
    void exceptionHandler1(...);
    void exceptionHandler2(...);
    ... ...
};

Class sftFramework involves a set of control modes corresponding to different schemes. For a given application, the programmer can choose appropriate control modes to realise particular fault-tolerant structures which could be the most effective for the application. The enum structure, sftStatus, enumerates possible execution states of a specific execution mode,
returned by the corresponding operation, including those such as NORMAL (a normal service), EXCEPTION (a possible degraded service) and FAILURE (without any service).

Mode 1: *Conditionally Sequential Execution*: For example, the recovery block scheme. The control program first saves the state of the system to permit backward error recovery, then invokes Variant 1 (through Pointer pv1) and calls the adjudicator (e.g. an acceptance test, through the pa pointer); and if this variant fails the adjudication check, restores the state of the system and invokes another variant.

Mode 2: *Full Parallel Execution*: NVP, NSCP and τ(n-1)-VP are typical examples. The control algorithm invokes all the variants and coordinates their execution through a synchronisation regime. It then calls the adjudicator. If no single, presumably correct result is determined, an exception is signalled to the higher level of the system.

Mode 3: *Fully Sequential Execution*: the control architecture executes all the variants sequentially and then invokes the adjudicator for the selection of a correct result. Sequential NVP [GAA80] and the certification trail scheme [SM90] are two such instances.

Mode 4: *Dynamically Adaptive Execution*: The control program organises the execution of variants in a dynamic fashion. It always invokes a so-called currently active set of variants and then calls the adjudicator. Once a correct result is obtained, it stops any further action immediately. If the current execution fails the adjudication check, it configures another set of variants to execute. The SCOP method [BDX92, XBD93] employs such a structure.

Other modes are possible and, by inheritance, they can be defined and implemented by the user as appropriate. However in general of the details of the control architecture will be hidden from the application programmer. This makes it possible to write fault-tolerant programs in a terse, but disciplined style, thereby benefiting reliability. Our framework is not dependent on any concrete implementation details, but the definition of the recovery block control mode, as an example, would presumably take something like the following form:

```c
sftStatus sftFramework::recoveryBlock (adjudicator* pa,
  variant* pv[],
  int n,...)
{
  for (i=0; i<n; i++)
  { pv[i]->variantDefinition();
    if (pa->acceptanceTest() == 1)
    { return 'NORMAL';}
  }
  return 'ERROR';
};
```

This example does not obviously deal with the state restoration problem. The literature [CR93] suggests several implementation schemes for state restoration in C++.

To summarize, Figure 2 explains how the object-oriented construction works and how application programmers can define their own class hierarchies by inheritance. Clearly, the control architecture implemented by Class sftFramework exercises its control through two abstract base classes, rather than using the concrete variant objects directly. This technical detail is important and it makes our construction, at least notionally, very generally applicable.
3.3. Use of the Fault Tolerance Schemes

Based on the proposed construction, what the application programmer is left to do in order to create a particular fault-tolerant software structure is:

(a) develop redundant variants of the parts of the overall program which it is desired to make fault-tolerant (ideally, if a library of variants corresponding to the special application exists, just select a set of variants from the library);

(b) select a basic adjudication function provided by the system or, if necessary, define a new adjudication function, and

(c) choose an appropriate control mode effective for the particular application requirements (reliability, performance, timeliness, efficiency etc.).

These are then grouped together, in a user-defined module, as illustrated below (for the case of using a recovery block structure).

```c
sft_module()
{
    ...
    sftFramework* pf;
    acceptanceTest* pa; //user-defined AT test
    variant1* pv[1] = &primary; //user-defined primary
    variant2* pv[2] = &alternate; //user-defined alternates
    ...
    status = pf->recoveryBlock(pa, pv[ ], ...);
    ...
};
```

If the programmer wishes to use the NVP approach instead, he can simply produce the following code in much the same way to the above.
sft_module()
{
    ... ...
    sftFramework* pf;
    exactVoting* pe;  //user-defined voter
    variant1* pv[1] = &version1;  //user-defined versions
    variant2* pv[2] = &version2;
    variant3* pv[3] = &version3;
    ...
    ...
    status = pf->nVersionProgramming(pe, pv[ ], ...,);
    ...
};

Obviously, other existing structures for software fault tolerance can be expressed in similar ways. The object-oriented construction of the framework is simplicity itself: following the declaration part, the required scheme is activated using a single statement.

4. Fault Tolerance in Concurrent Object-Oriented Programming

The object-oriented framework for supporting software fault tolerance proposed in Section 3 is generally applicable in sequential programming. An extension, essential for such application domains as real-time processing, operating systems and distributed computation, is to support concurrent system recovery as well. Ideally, the fault tolerance problems of both sequential and concurrent object-oriented programming can be handled in a single framework. In traditional approaches based on communicating processes, the conversation [Ran75] provides a general solution to backward error recovery of concurrent systems. Unfortunately, a similar standard structuring concept cannot be obtained by just incorporating the conversation scheme into concurrent object-oriented computing. Many new problems and difficulties arise when object-oriented programming is considered. In this section, we describe these problems and present a solution by building an appropriate abstract framework.

4.1. Difficulties

First of all, it is essential to obtain a standard linguistic framework for enforcing software fault tolerance in the context of concurrent object-oriented programming, which might be used in a practical language. However this is difficult since, on the one hand, mainstream object-oriented languages like C++ and Eiffel do not at present address concurrency and, on the other hand, a large number of quite different models for concurrent object-oriented programming have been proposed but none has yet received widespread acceptance. These models use different entities as the building blocks of concurrency, including tasks, processes, active objects, asynchronous operation execution and threads. So, previous syntax proposals for software fault tolerance in concurrent systems [Kim82, GK89] based on the traditional communicating process model can provide little help to the implementation of concurrent error recovery in a practical object-oriented language.

In general, there are essentially two basic strategies for introducing concurrency features into the sequential object-oriented programming paradigm. One approach is to superimpose the concurrency constructs as an additional layer orthogonal to the object-oriented programming features, while the other approach attempts to achieve full integration at the same level. The orthogonal approach is used, for example, in Concurrent C++ [GR88]. The concurrent structure of Concurrent C++ is completely independent of the encapsulation and
communication mechanisms of the objects in a system. Because its concurrent structure is based on the communicating process model, the conversation concept and the related syntax can be exploited directly. But the approach has the major disadvantage that programmers are required to work at two different levels of abstraction, and thus is dangerously error-prone.

Many other object-oriented languages designed for the purpose of supporting concurrency adopt the integration approach which is much more in keeping with the general goal of unifying different concerns through the notion of objects. However, there are quite a number of different ways in which this goal can be achieved. Two of the most basic techniques for generating multiple execution threads are active objects and asynchronous operation execution.

The active object approach (e.g., as in POOL and Concurrent Smalltalk [Ame87,YT86]) regards an object as an active process and associates a permanent thread with the object as a whole. Alternatively, in the second approach (e.g., as in Hybrid and the Actor languages [Nie87, Agh86]), a new execution thread is generated to execute the body of an operation in response to an invocation request (message). The client object, therefore, is free to proceed (asynchronously) with the execution of the operation as soon as the call is accepted. The fundamental difference between these two approaches is essentially one of granularity of concurrency. The latter offers much finer grained concurrency than the former, since it supports parallelism at the level of individual operations by generating a separate execution thread for each operation invocation rather than for a set of operations associated with a given object. Each is appropriate for certain application domains. The difficult aspect, however, is to deal with the fault tolerance problem of these two models within a unified framework. Appropriate abstractions must be built.

4.2. Building High Level Abstractions

A unified framework intended to support concurrent system recovery necessarily concerns three main aspects: concurrent execution threads, inter-thread communication and error recovery of threads. We here provide a preliminary sketch of a unified treatment of different concurrent models and a simple framework in relation to these three aspects below.

**Concurrent execution threads:** In our framework, a thread can be a process as used in the traditional CSP model [Hoa85], which may be an instance of a process (task) type (e.g., as in Concurrent C++) and be active, having a script of its own to execute. It also can be regarded as an active object. In fact, there are strong similarities between processes and active objects. An active object has its own script or body plus a set of operations. After its creation, the active object executes its body. When the routine that describes the script terminates, the execution thread terminates. In the 'synchronous execution operation' model, the operations applicable to an object may be viewed as its scripts; the difference is that an object may have more than one script. Thus, a single operation forms an execution thread in this model.

**Inter-thread communication:** Basically, inter-thread communication in our framework could be based on the traditional message passing mechanism. In object-oriented programming, however, the situation is different since communication is already present as a part of fundamental operation -- an operation call (e.g., target.operation(arg1,...) denotes the application of operation to the object to which target is attached). This is actually a communication action. In the 'asynchronous operation execution' scheme, because a
client proceeds asynchronously with the execution of an operation, it is not possible, in a single transaction, to transfer information from the server to the client on completion of the operation body. Such transactions, therefore, reduce to the simple unidirectional message passing model. If necessary, any required data exchange from the server to the client has to be programmed as a separate transaction. Although asynchronous message passing of this kind can be adopted in the active object model of concurrency, most object-oriented languages adopting the active object approach view the operation call as a form of 'remote procedure call' mechanism, involving two-way communication and interposed code execution. Our framework, therefore, involves two forms of communication -- asynchronous message passing and synchronous remote procedure call. Note that we will not particularly deal with synchronization in the framework since it is not much related to our concern here and can be regarded as a special case of communication.

**Error recovery**:Forward error recovery can be associated with concurrent execution threads [CR86], but major problems arise when backward recovery and alternate execution threads are considered. On entry to a conversation (or more generally a recoverable atomic action), a thread establishes a recovery point and, thereafter, may only communicate with others that have also entered the action. If a thread needs to recover while in an atomic action, then all other threads of that action are also forced to recover and each may execute an alternate algorithm. Moreover, when a thread wishes to leave the action, it must wait until all other threads are ready to leave. In the communicating process model, an alternate execution thread can be a new process or an alternate execution segment of the original process. Being similar, the alternate code can be involved in the script of an active object or be another active object that may have different script and operations. When the execution thread is a single operation, its alternate is another operation. But the alternate operation is not necessarily applicable to the same (original) object, as will be further discussed below.

### 4.3. Construction of Concurrent Error Recovery

We will now describe the object-oriented construction of the abstract framework outlined in the last subsection. In the construction, atomic actions are explicitly separated from some backward recovery activities. This separation is motivated by the following reasons.

1. Separating atomicity and backward recovery is particularly helpful in controlling the complexity of constructing a recoverable atomic action (or a conversation). An atomic action in our construction fulfills just three obligations: (i) asynchronous entry (which may involve creating appropriate recovery points), (ii) prohibition of information smuggling (namely, no thread is allowed to communicate with a thread that is not participating in the action), and (iii) synchronous exit (which may involve discarding recovery points or restoring the states from those recovery points at entry). Some forms of forward recovery can be associated with those operations at the exit, but major activities of backward recovery, including organization and invocation of the alternate execution threads and failure notification while alternates are exhausted, are dealt with somewhere else (e.g., our construction handles backward recovery through a subclass of the sftFramework class). The functionality of an atomic action, therefore, is well defined.
(2) This separation permits truly independent alternate algorithms to the extent that a thread can communicate with different groups of threads to achieve its goals and thus supports great design diversity. In our construction, should an atomic action fail, an alternate action can be a quite different action that may contain a entirely separate group of execution threads.

(3) Although our atomic action does not deal with backward recovery, it still provides strong support for subsequent recovery activities. The state preservation and restoration enable backward recovery, while the prohibition of smuggling enforces effective recovery and limits the distance of rollback.

Below we provide a class for facilitating the organization of atomic actions. The class is named ftAction (i.e., fault-tolerant action) in order to indicate differences between it and other atomic-action abstractions [SDP91].

```c++
enum actionStatus {SUCCESS, FAILURE...}
enum checkStatus {ALLOWABLE, PROHIBITED, ...}

class ftAction
{
    ... ...
    //private variables and operations
    //the class invariant

    public:
    virtual void entryAction(thread_id,...);
    virtual checkStatus atomicity(thread_id);
    virtual actionStatus exitAction(*acceTest,...);
    ... ...
};
```

An object of type ftAction specifies a unique atomic action. It is common to those threads which are or will be participating in the action. Two enum structures, actionStatus and checkStatus, are defined: one enumerates possible result states of an action and the other indicates the conclusion of an atomicity check. The entryAction operation causes the thread specified by thread_id to enter an action and creates a recovery point. The function of operation atomicity is to enforce atomicity of communication. When a thread in the action wishes to communicate with another thread specified by thread_id, it must first call the atomicity operation which checks if the target thread is participating in the same action. The exitAction operation is actually a form of synchronization operation. Each thread may be associated with a local acceptance test specified by the acceTest pointer. When it arrives at the exit (i.e., calls the exitAction operation), the test is evaluated. If it passes the acceptance test, it becomes ready to leave. However, each thread in the same action must wait for all the others to arrive at an exit. If all the threads are ready to leave, a global acceptance test may be applied. Once that the global test is passed, all the recovery points are discarded and the action ends successfully. Should any thread fail its acceptance test (or any other failure of one of the threads occur) or they fail the global test, the action fails and all effects since the entrance to the action are undone by restoring the states. An appropriate exception is signalled.

The basic operations in class ftAction are all declared as virtual so that the programmer can redefine them for additional functionality requirements. For example, the atomicity operation
can be extended to check the incoming information such as operation calls and messages passed by other threads. Besides, as mentioned above, forward recovery can be incorporated into the exitAction operation. Once an error is detected, appropriate forward recovery measures would be invoked.

Concurrent backward recovery can be provided by simply writing a subclass, named csftFramework, of the sftFramework class so as to extend the construction used for sequential programs to concurrent programming and, again, to hide the control details from the application programmer. The csftFramework class defines new control modes, including the concurrentBackwardRecovery operation:

`public: sftStatus concurrentBackwardRecovery(ftAction A[], ...);`

Since the operation of concurrent backward recovery may be related to a group of fault-tolerant actions, an object of type csftFramework should be common to all the actions involved in the recovery activities. The operation corresponds to a control mode: first attempt a primary action, say ftAction A[0], and wait for completion of the action — 'SUCCESS' or 'FAILURE' (the status information returned by the exit operation when the action ends). If the action is successful, the operation is completed normally and control proceeds with the statements following the concurrentBackwardRecovery operation. Should the action fail, an alternate action, say ftAction A[1], is attempted, and so on sequentially until either an action ends successfully or all the attempts fail.

From the perspective of each thread, the set of fault-tolerant actions in which it participates actually constitutes its primary and the series of alternates of a backward recovery structure. An example for a particular thread threadT that exploits concurrent recovery is as follows. Note that, though not shown in the below example, an atomic action itself can be nested to support a hierarchy of recovery regions.

```c
    csftFramework* pc;
    ftAction A, B;
    ...
    threadT()              //a particular execution thread
    {
        ...
        status = pc->concurrentBackwardRecovery(A,B,...);
        ...
        A.entryAction(id,...);
        ...
        //action A
        if (A.atomicity(target_id) == 'ALLOWABLE')
        { //communication with the target thread
            ...
        } else ... //an error
        ...
        ...A.exitAction(pa,...);
        ...
        ...
        B.entryAction(id,...);
        ...
        //action B
        ...B.exitAction(pb,...);
        ...
    }
```

At execution time, when control of the threadT thread reaches the statement of Operation concurrentBackwardRecovery, the thread will attempt to perform the activities represented within ftAction A. If the attempt is successful (i.e., A.exitAction(pa,...) returns 'SUCCESS'), control proceeds with the code following the recovery operation; otherwise the B action will be tried. Exhaustion of all the attempts for the threadT thread without success will
lead the recovery operation to return the status information -- 'FAILURE'. Although communication atomicity is enforced, the thread may communicate with entirely different sets of threads in each action, thereby allowing great diversity between the alternate actions. Also, each thread may specify a different number of alternate actions from the other threads to accommodate its own goal. Note that, for effective and correct operation, the application programmer is required to adhere to a set of programming conventions still. But we would claim that such conventions in our construction are standard, systematic and simple. Apart from the necessary entry and exit conventions, the atomicity check is very much like the use of a standard precondition in Eiffel.

5. Conclusions and Discussions

The contributions of the work reported here include

(1) a coherent framework for enforcing software fault tolerance and an object-oriented construction of the framework that help to produce compact dependable and fault-tolerant software in a disciplined manner;

(2) a demonstration of the way in which this framework can facilitate the selective use, both singly and in combination, of a variety of existing software fault tolerance strategies;

(3) a better understanding of different concurrent object-oriented programming models through high level abstractions and a further unification of them based on the generalized notions of threads, inter-thread communication and error recovery, allowing a unified framework for incorporating fault tolerance into concurrent object-oriented systems.

The C++-like notation used in the report does not limit the applicability of our methodology. The object-oriented construction of our framework can be implemented using any language that supports data abstraction, inheritance and polymorphism. However, depending on the actual language chosen, use of the framework may depend to some degree on the programmer adhering to a set of programming conventions, though a pre-processor could be used to enforce these conventions. (For example, the programmer might have to include explicit calls in each operation of an object to facilities related to the provision of state restoration.) It would however be preferable if such conventions could be made an integral part of the language used by the application programmer.

The concept of reflection [MN87] could be highly relevant here, since the combination of reflection and object-oriented programming in the form of a meta-object protocol [AFPS92] makes it possible to change, for the duration of a given scope, the way in which a particular program is interpreted. For example, such actions as object creation/destruction and method invocation could be temporarily and transparently enhanced. Thus what would otherwise have depended on the programmer adhering to particular conventions could instead be automatically invoked on his behalf. Such possibilities however need further investigation.

We have not discussed in detail the problem of state preservation and restoration, which is dealt with in [CS93]. Several techniques that this report describes can be directly applied to our

Page 19
construction. We have also ignored the conflict between maximum concurrency and controlled recovery. Synchronous exits from an atomic action could defeat the purpose of concurrent computation, but such a synchronization overhead is the price paid for controlled recovery.

6. Acknowledgements

The work described in this paper has benefited greatly from discussions with a number of colleagues in the PDCS2 project, in particular Cecilia Rubira-Calsavara and Robert Stroud.

References


