Software Fault Tolerance: Towards an Object-Oriented Approach

J. Xu, B. Randell, C.M.F. Rubira-Calsavara and R.J. Stroud
Software Fault Tolerance: Towards an Object-Oriented Approach

J. Xu, B. Randell, C.M.F. Rubira-Calsavara and R.J. Stroud

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. The additional software redundancy may increase
the size and complexity and thus adversely affect software reliability.
Object-oriented programming seems to provide an appropriate framework
for controlling complexity and enforcing reliability. However, software
fault tolerance cannot be achieved merely by implementing the classical
fault tolerance schemes in an object-oriented fashion. New problems arise
while integrating software redundancy into object-oriented computing
systems. This paper identifies a set of such problems, addresses possible
solutions and proposes an object-oriented architecture for dealing with
software design faults. Both linguistic supports for the architecture and
implementation issues are also discussed in detail.
Bibliographical details

SOFTWARE FAULT TOLERANCE: TOWARDS AN OBJECT-ORIENTED APPROACH.

Software Fault Tolerance: Towards an Object-Oriented Approach.
[By] J. Xu [and others]

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no. 498)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
XU, Jie

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. The additional software redundancy may increase the size and complexity and thus adversely affect software reliability. Object-oriented programming seems to provide an appropriate framework for controlling complexity and enforcing reliability. However, software fault tolerance cannot be achieved merely by implementing the classical fault tolerance schemes in an object-oriented fashion. New problems arise while integrating software redundancy into object-oriented computing systems. This paper identifies a set of such problems, addresses possible solutions and proposes an object-oriented architecture for dealing with software design faults. Both linguistic supports for the architecture and implementation issues are also discussed in detail.

About the author

Mr. Jie Xu joined the Department of Computing Science at the University of Newcastle upon Tyne in April 1990, where he is a Research Associate.

Professor Randell has been a Professor of Computing Science in the Department of Computing Science at the University of Newcastle upon Tyne since 1969.

Cecilia M. F. Rubira-Calsavara joined the Department of Computing Science at the University of Newcastle upon Tyne in 1990, where she is a PhD student.

Dr. Robert J. Stroud joined the Department of Computing Science at the University of Newcastle upon Tyne as a Research Associate in 1983 and has been a Lecturer since 1991.

Suggested keywords

ABSTRACTION          OBJECT-ORIENTED PROGRAMMING          PROGRAM STRUCTURING
REFLECTION            SOFTWARE FAULT TOLERANCE

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.6425          U.D.C. 681.322.06
Software Fault Tolerance: Towards an Object-Oriented Approach

J. Xu, B. Randell, C. M. F. Rubira-Calsavara and R. J. Stroud

University of Newcastle upon Tyne

Newcastle upon Tyne, UK

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. The additional software redundancy may increase the size and complexity and thus adversely affect software reliability. Object-oriented programming seems to provide an appropriate framework for controlling complexity and enforcing reliability. However, software fault tolerance cannot be achieved merely by implementing the classical fault tolerance schemes in an object-oriented fashion. New problems arise while integrating software redundancy into object-oriented computing systems. This paper identifies a set of such problems, addresses possible solutions and proposes an object-oriented architecture for dealing with software design faults. Both linguistic supports for the architecture and implementation issues are also discussed in detail.

Key words— Abstraction, object-oriented programming, program structuring, reflection, software fault tolerance.

This paper has been published in IEEE FTPDS-94, College Station, Texas, June 1994 and will appear in the book Fault-Tolerant Parallel and Distributed Systems published by IEEE Computer Society Press.
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 (see for example the set of papers in [Voges 1988]). In what follows, we shall take some canonical proposals and methods as examples to explain software fault tolerance concepts — we will not concern ourselves with the techniques designed to tolerate the effects of faults in hardware which happen to be implemented in software. Strigini presented in [Strigini 1990] a comprehensive survey of software fault tolerance issues, where further references can be found.

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 the 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. The purpose of this paper is to explore the application of some new ideas in program structuring, and particularly in object-oriented programming [Meyer 1988], to the provision of software fault tolerance in the hope that redundancy can be incorporated in software in a disciplined and modular way and that the impact on system complexity could be controlled and minimized.

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 other aim is to show how these forms of software fault tolerance could be provided in general purpose languages and within a standard architecture by the use of some of the newer techniques in object-oriented programming, such as reflection and meta-level programming [Maes 1987].

However, software fault tolerance cannot be obtained merely by implementing or programming the existing software fault tolerance schemes in an object-oriented manner. New problems emerge and some trivial problems in conventional (function-oriented) programming become dominating when object orientation is considered. To our knowledge, few researches have explored the potential benefits and possible problems of using object-oriented techniques to facilitate design fault tolerance and to deal with design diversity. In this paper, we will demonstrate our effort towards an object-oriented approach to enforcing software fault tolerance. The paper is organized as follows. Section 2 discusses the problem of how to incorporate software fault tolerance into (object-oriented) systems in a disciplined and well-coordinated fashion. Section 3 identifies new problems while considering object-oriented software fault tolerance and suggests possible solutions. An object-oriented architecture for describing various forms of software fault tolerance is provided and a linguistic instance of this object-oriented architecture is produced based on a set of pre-defined virtual classes. Section 4 deals with the implementation issues of the object-oriented construction, both linguistic and mechanistic ones, and explores the usefulness of the reflection technique for these issues. The final section presents conclusions.
2 System Structuring and Software Fault Tolerance

In order to discuss software fault tolerance, we must first establish or obtain an abstract model of describing software systems. A system is defined to consist of a set of components which interact under the control of a design [Lee and Anderson 1990]. The components themselves may be viewed as systems in their own right. In particular, the design of a system is also a component, but has special characteristics such as the responsibilities for controlling the interactions between components and determining connections between the system and its environment.

2.1 Idealized Components

An idealized component 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 on system complexity [Anderson 1985][Lee and Anderson 1990]. The above part of Fig.1 shows such an idealized component. Fault tolerance is here obtained by exception handling without the use of diverse designs. Exception handling is often considered as being a limited form of software fault tolerance; for example, by detecting and recovering an error, and either ignoring the operation where the fault manifested itself or by providing a pre-defined and heavily degraded response to that operation. In some sense, the software 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 which is robust in the sense that catastrophic failure can be averted.

2.2 Software Fault Tolerance Schemes

The redundancy required to be able to tolerate software faults is not just 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. Masking redundancy, also known as “static redundancy”, uses extra software components of diverse design (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 [Avizienis and Chen 1977], a newer one being i/(n-1)-variant programming [Xu 1991][Xu and Randell 1992]. 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.

A system with dynamic redundancy consists of several redundant components with just a subset, typically one, active at a time. If a software error is detected in the active component it is replaced by a spare component. Three examples of the use of dynamic redundancy are recovery blocks [Randell 1975], n-self checking programming [Laprie et al. 1987], and self-configuring optimal programming [Bondavalli et al. 1993][Xu et al. 1993]. 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.

2.3 Idealized Components with Diverse Design

Incorporation of (true) software fault tolerance in systems requires a structured and disciplined approach. The concept of an idealized component is not directly applicable. We need a simple abstraction model to describe common characteristics of the existing software fault tolerance schemes. The below part of Fig.1 suggests an abstraction model (or architecture) of a fault-tolerant software component and shows the details of the component, called controller, which contains multiple sub-components: redundant variants of diverse design and an adjudicator [Anderson 1986]. 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 controller controls the execution of the variants and determines the overall system output with the aid of the adjudicator.

![Diagram of Idealized Component with Diverse Design]

**Fig.1 Idealized component with diverse design**

Our concept of an idealized component with diverse design is a natural extension and generalisation of that of the idealized component, adhering to the same external characteristics as those that an ideal component exhibits and combining exception handling and design fault tolerance within a unified framework. In particular, the design of the component is embodied in the controller (the control algorithm) which invokes one or more of the variants, waits for the
variants to complete their execution and then invokes the adjudicator to perform a check on the results of the variants. Following the idea of recursive structuring, a subcomponent (e.g. a software variant) can further use various fault-tolerant techniques providing it keeps the external characteristics of an idealized component.

In fact, the whole organization is in principle fully recursive. Each of the variants, the adjudicator and the controller in our model itself is an idealized component and may have a set of exception handlers associated with it. Like the exceptional situation of the controller component shown in the above part of Fig.1, in a variant which is a sub-component of the controller, for example, 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 controller that it has been unable to provide the service requested of it.

2.4 Object-Oriented System Structuring

The system structuring discussed above reflects a traditional functional view of software design. However, the abstraction architecture is equally appropriate for object-oriented programming. In fact, the object-oriented paradigm fits closely with the idea of idealized components. An ideal fault-tolerant component, or in general a component, can conveniently be thought of as an object [Lee and 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. In particular, the object-oriented approach emphasizes the use of classes and inheritance — an object is an instance of some class or type. Based on the concept of abstract data type, it is natural to describe those components in our model in terms of three distinct classes, respectively corresponding to variants, adjudicators and controllers, and this is taken as a possible program implementation example in Section 3. In the example, we regard the classes 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 could be many other ways of implementing software redundancy in an object-oriented fashion. However, we have found it is not an easy job to achieve software fault tolerance under the paradigm of object-orientation. Many new problems arise.

3 Design Redundancy and Object-Orientation

3.1 Granularity of Redundancy in Object-Oriented Programming

Redundancy of design can be incorporated into object-oriented programming at (at least) three different levels of granularity: 1) individual operations (or methods) or part of an operation, 2) different objects (from the same class), and 3) different classes (i.e. different objects from different classes).
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, the existing techniques and experiences in conventional programming can be employed most directly. For example, error recovery can be naturally done by restoring all modified non-local variables. (Note that the situation becomes much more complex in concurrent systems.) In practice, further decisions need to be made regarding the implementation of this kind of operation. Different degrees of transparency could be provided for the user of the special class — from full transparency as a normal operation (the 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: Design fault tolerance can be achieved by diversity in the data spaces 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 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 and Knight 1987] showed data diversity can be effective and very economical. Of course, redundancy at object-level can 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. There are two similar approaches to introducing redundancy of design at 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 example in Subsection 3.3 employs this approach), or the variants are declared as different classes and regarded as different implementations to a given type (see for example the approach used in the Arche system [Benveniste and Issarny 1992]). Although this strategy seems to be the best choice, further problems arise, especially in the state saving and restoration — we will discuss these problems in the next subsection.

Granularity of redundancy can be further enlarged to the meta-level, or the system-level redundancy depending upon special application requirements. For example, the application of n-version programming is often limited to the outermost, system, level of the software though there is no conceptual reason why the n-version scheme cannot be applied at both system and component (object or class) levels.

3.2 Software Variants as Objects and State Restoration

In the masking redundancy schemes, such as n-version programming, all variants are normally executed while 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 the variants and reduces the data which must be passed to a variant upon invocation. However, the recovery block approach with dynamic redundancy is different
and does not execute all the variants each time unless it is necessary. Therefore 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 upon recovery. There are several solutions of the problem. The variants in recovery blocks are designed 1) as memoryless functional components rather than objects, i.e. diverse design is limited to the operation-level; 2) as special objects which do not retain local data or only retain a limited amount of local data, or 3) 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.

However, although all the variants could be executed upon each invocation, further problems will emerge if the 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 failures will eventually exceed the fault-masking-ability, and the entire fault-tolerant system will fail. Here, the problem becomes that of how to perform error recovery while keeping design diversity in mind.

One method of conducting recovery would be for each variant to roll itself back to the state that it was in prior to its last operation, i.e. to produce no result. Since each variant 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 faulty variant to one which corresponds to those of the other up-to-date variants. This can easily be done if the variants have the same internal data structure, such as the community error recovery method in [Tso and Avizienis 1987]. Recovery is difficult while supporting full design diversity — these objects may be independently designed and their internal data structures will, in general, be different. The mapping relationship between the internal state of one variant and that of another is the key and must be obtained.

Tso and Avizienis [Tso and Avizienis 1987] argued the constraints on full design diversity that result from requiring the same or similar internal data structures in the variants for 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 tradeoff 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.

3.3 An Example: Pre-Defined Classes for Software Redundancy

Our abstraction architecture of an idealized component with diverse design provides an implementation framework for software fault tolerance. The major advantage of this framework is to facilitate the flexible, selective use, both singly and in combination, of a variety of existing software fault tolerance strategies. In this subsection, we will briefly describe a possible object-oriented implementation by introducing several special classes — more technical program
details could be found in [Randell and Xu 1993]. We use a (C++)-like notation [Stroustrup 1991] to express the object-oriented construction of the fault-tolerant component in order to demonstrate how software fault tolerance could be provided in mainstream languages, though our classes can also be simply described in other languages. Our classes are to be built on top of a language such as C++ or Eiffel [Meyer 1992] without modification of the compiler, so that rapid and instructive experiments are possible. Implementation issues will be further discussed in the fourth section where other possible implementations are evaluated.

We first introduce the \texttt{variant} class which characterizes a variant component:

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

    ...
}
\end{verbatim}

The function of the pre-defined \texttt{variant} class is to provide an interface through which the application programmer can develop particular-concrete-program variants (e.g. variants of a sorting operation) and, furthermore, the controller can organise the execution of these user-defined variants so as to provide either static or dynamic redundancy, as specified by the application designer. Class \texttt{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 \texttt{variantDefinition()} function that is intended to be overridden in the derived (and user-defined) classes. (For \texttt{virtual} functions, it is possible in C++ to declare a function without providing a definition.) Polymorphism is implemented via \texttt{virtual} functions: the application programmer defines these \texttt{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, \texttt{sortVariant1}, in the form of a derived class.

\begin{verbatim}
class sortVariant1 : public variant
{
    int array[]... //private attributes

    //interface to controller
    int variantDefinition(int);

    //user-defined sorting
    void sort(array[])

    //user-defined handlers
    void exceptionHandler1(...);
    void exceptionHandler2(...);

    ...
}
\end{verbatim}

Clearly, there is no essential difference between the special class used as a variant in a fault tolerance scheme and a general-purpose class used in non-fault-tolerant applications. 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. It is worth noting that a
programmer who wishes to invoke a sort operation on this object does not have to write variantDefinition instead of a more natural name like sort. A simple way of arranging this is to write the variantDefinition operation as a caller of a or several concrete operations, such as the sort operation.

A basic set of adjudication algorithms (e.g. majority voting based on equality checks, and simple reasonableness checks) would be provided by the system. However, in practice, the adjudication algorithms are often application-specific; they can thus be a user-provided error detection measure and can be as reliable or as complicated as the application programmer wishes. Following our idea on the development of software variants, the application programmer can easily insert such a measure to a program. We now provide the adjudicator class below, again as an abstract base class.

Although it is hoped that the adjudicator can be kept simple, abnormal events might still occur during its execution. Exception handling measures could therefore be provided as part of the adjudicator. Moreover, it is possible for the adjudicator to receive an invalid result from a variant. 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 controller).

```
class adjudicator
{
    //private variables and operations
    public:
        virtual boolean acceptanceTest(); //standard adjudication
        virtual void voter(...); //operations
        ... ...
        void exceptionHandler1(...); //handlers for errors in
        void exceptionHandler2(...); //adjudicators
        ... ...
};
```

Finally, the function of the controller embedded within an ideal component that incorporates diverse design 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 algorithms would be provided by the system and can be specified by the application programmer.

```
enum sftStatus (NORMAL, EXCEPTION, FAILURE...)

class controller
{
    //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(...);
        ... ...
};
```

Here class controller involves a set of control modes corresponding to different software fault tolerance 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). 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 control algorithms 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. These pre-defined classes actually doesn't set any limitation to concrete implementation details, but the definition of the recovery block control mode, as an example, would presumably take something like the following form. (Notice that this example does not obviously deal with the state restoration problem — it may be treated in a different level of the supporting system as suggested in our reflective architecture of the next section or as described by the literature [Rubira-Calsavara and Stroud 1993] where several implementation schemes are discussed and provided for state restoration in C++.)

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

To summarize, Fig.2 explains how the object-oriented construction works and how application programmers can define their own class hierarchies by inheritance. The controller 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.

![Fig.2 Pre-defined classes and user-defined class hierarchy](image)

10
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; (b) select a basic adjudication function provided by the system or, if necessary, define a new adjudication function, and (c) specify an appropriate control mode. These are then grouped together, in a user-defined module, as illustrated below (for the case of using a recovery block structure).

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

If the programmer wishes to use the NVP approach or other fault-tolerant structures instead, he/she can simply produce the code in much the same way as above. The object-oriented construction is simplicity itself: following the declaration part, the required scheme is activated using a single statement. In the style of object-orientation, the application programmer can construct an object-oriented system in such a way in which the creation of a fault-tolerant object may actually correspond to the creation of a group of diverse objects. When the user of the fault-tolerant object invokes an operation, e.g. `sort()`, the operation which may contain the above code would cause the invocation of a group of diverse operations, e.g. `sortVariant1()`, `sortVariant2()` etc., on the respective variant objects.

4 Implementation Issues: Linguistic and Mechanistic Supports

Mainstream object-oriented languages such as C++ and Eiffel lack support for software fault tolerance. Our new classes introduced in the last section could be viewed as an extension of the C++ language to provide software redundancy. Their advantage is that no modification to the compiler is needed so enabling rapid experimentation. The serious problems of this method are intensive use of inheritance and polymorphism, special programming conventions and poor efficiency.

4.1 Problems and Possible Solutions

When software fault tolerance approaches are aimed at providing fault-tolerant functional components which may be nested within a large program, both linguistic and mechanistic supports are generally demanded. For example, the classical recovery block approach requires the basic program features: `ensure AT by M1 else by M2 ... else by Mn else error`, and a suitable mechanism for providing automatic backward error recovery. The simplest method for implementing recovery blocks would be to develop a set of guidelines to show how to use a chosen language to express and implement the functionality of recovery blocks, assuming that the language chosen provides enough expressibility. The application programmer who wishes to utilize software fault tolerance must strictly adhere to the guidelines, and all checks as to the adherence to them must be performed only by the
programmer himself. This is of course a fruitful source of software design faults and therefore defeat its own purpose of reliability improvement. The object-library method used in our example suffers from the similar problem, but reduces to some extent the extra burden of fault tolerance design, thereby decreasing the possibility of introducing new faults into program.

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 C++ 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 C++ program generated by the pre-processor, rather than of the extended C++ program that they had written [Randell 1993]. Besides, large amount of modification to the language makes immediate experiments of fault tolerance concepts difficult.

We would claim here that mechanisms for supporting design redundancy cannot be made totally transparent to the application programmer who wishes to utilize them. There is a fundamental difference between object replicates and diverse design of objects. The use of the former could be made transparent to the programmer and performed automatically by a supporting system. However, it has to be the responsibility of the application programmers for developing software variants, acceptance tests, and even the application-specific voters. Special language features and/or programming conventions therefore cannot be avoided completely. In consideration of software reliability, the key problem would become how a set of simple (thus easy to check) programming features can be developed with powerful expressibility and how the supporting mechanisms such as those for state restoration can be provided in a more natural and modular manner rather than by an ad-hoc method such as system calls.

Suppose that a set of software variants are designed as a group of objects which are respective instances of different classes. To facilitate software fault tolerance, two special features or constructs would be helpful and necessary: a construct for the declaration of a group of objects (or object variants) and an extension of the semantics of operation calls to the invocation of a object group. The Arche language [Benveniste and Issarny 1992] declares the object group as a sequence via the type constructor SEQ OF and provides the multi-operation facility which supports operation calls on an object group. Clearly, this new language can simplify the expression of such a software fault tolerance scheme as n-version programming. However, it does not provide powerful expressibility to enable the implementation of various types (static or dynamic) of software fault tolerance within a unified framework since the semantics of multi-operations is statically fixed. What we desire is that the extended semantics of operation calls can contain different control modes which would correspond to sequential, or adaptive, or concurrent invocation of the object variants. Note that if no diversity is involved in the design of an adjudicator, the expression and implementation of it will be relatively easy and it could be declared as a normal object or operation.
4.2 Reflection and Reflective Architecture

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 (the meta-level representation) of the system, and it also provides powerful expressibility and encourages modular descriptions of computation by introducing a new dimension of modularity — the separation of object-level descriptions and meta-level descriptions. 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. In our case, 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.

Our abstraction model (or architecture) for software fault tolerance helps the separation of object-level and meta-level descriptions. At object-level (or base-level), just linguistic supports for object groups and multi-object operations would be required. The controllers which control the execution of object variants are naturally implemented as meta-objects. The actual execution of an operation call on an object group is controlled and dynamically reified at meta-level. Since a meta-object is also an object, it can be controlled by a meta-meta-object. The meta-level operations (e.g. the fault-tolerant controllers) can be further reified at meta-meta-level, containing a provision of supporting mechanisms for state restoration and synchronization and so on. Such an ascending tower of meta-objects enforces multi-level modularity and facilitates dynamic change in the system behaviour. Figure 3 illustrates our reflective architecture for software fault tolerance based on our model of fault-tolerant components described in Section 2.

![Diagram](image)

**Fig. 3 Reflective architecture for software fault tolerance**

Let us now suppose that a robust set of data is defined as a group of diverse objects, called $G$, with the same type $\text{robustData}$. The $\text{robustData}$ type provides an operation $\text{int getMin()}$ which returns the smallest integer in the set of data. A call to $\text{getMin()}$ on $G$ would have the form
\( m = G.\text{getMin()} \)

This operation call would then be reified at meta-level following the appropriate control mode, say, the recovery block mode. That is, the \text{getMin()} operation would be applied on the objects in \( G \) sequentially, depending upon error detection. An acceptance test can be easily passed onto the meta-level through an input parameter of the operation \( \text{getMin}(\text{adjudicator a}) \)

This mode is further reified at the higher meta-level to involve automatic state restoration upon recovery. When the need arises, the recovery block mode at meta-level can be dynamically switched over to the NVP mode or other modes even at runtime. Although meta-code at meta-levels is usually written only by a system specialist, the application programmer should be able to specify the desired control mode or to incorporate new meta-code into meta-levels whenever it is necessary for certain applications.

In practice, we need a language that is both popular and reflective. The present ANSI proposal for C++ does not consider the support for the reflection concept. [Chiba and Masuda 1993] describes an extension of C++ to provide a limited form of computational reflection, called Open C++. Our reflective architecture for software redundancy is being under evaluation and experimentation based on Open C++. Preliminary results are promising though some inflexibilities and rigidities exist.

5. Conclusions

We have extended the idealized component concept to the incorporation of software design redundancy. The extended model suggests a coherent framework for enforcing software fault tolerance and helps to produce compact dependable and fault-tolerant software in a disciplined manner. New problems are identified and addressed when object-oriented software redundancy is considered. A set of pre-defined classes demonstrates the way in which the proposed framework can facilitate the selective and flexible use, both singly and in combination, of a variety of existing software fault tolerance strategies. Obviously, the notion of ideal components and systems needs to be used in such a way as to achieve an appropriate structuring of the complex asynchronous activities to which the system can give rise, in particular those related to fault tolerance (e.g. the use of atomic actions).

Depending on the actual language chosen, use of the proposed architecture may depend to some degree on the programmer adhering to a set of programming conventions, though a preprocessor could be used to enforce these conventions. The meta-level technique is particularly useful in making our architecture reflective. For example, as showed in the later sections of the paper, 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 power and usefulness of reflective capabilities however need further verification and evaluation, especially in complex concurrent systems.
6 Acknowledgements

This research was supported by the CEC-sponsored ESPRIT Basic Research Action on Predictably Dependable Computing Systems (PDCS and PDCS2).

References


