Toward an Object-Oriented Approach to Software Fault Tolerance

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

University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK

In: **Fault-Tolerant Parallel and Distributed Systems**,!
Avresky, D.R. (ed.)

IEEE Press,! ISBN:!0-792-38069-X,! 1994

© 1994 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Toward an Object-Oriented Approach to Software Fault Tolerance

J. Xu, B. Randell, C. M. F. Rubira-Calsavara and R. J. Stroud
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK

Abstract

Software fault tolerance is often necessary, but can itself be dangerously error-prone because of the additional effort that must be involved in the programming process. The additional redundancy may increase size and complexity and thus adversely affect software reliability. Object-oriented programming provides 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 discussed in detail.

1: Introduction

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 [10]. In what follows, we shall take some canonical proposals and methods as examples to explain software fault-tolerance concepts. Strigini presented a comprehensive survey of software fault-tolerance issues in [9], where further references can be found.

Software design fault tolerance is often necessary, but can be error-prone — redundancy of design 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. We explore in this paper the application of some new ideas in program structuring, and particularly in *object-oriented programming*, to the provision of software fault tolerance in the hope that redundancy can be incorporated into software in a disciplined way and that the impact on system complexity can be well controlled.

The various forms of software fault tolerance that have been devised to date either require special language features (provided by a special compiler or preprocessor) 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 uniform architecture by the use of some of the newer techniques in object-oriented programming, such as *reflection* and *meta-level programming* [6].

Software fault tolerance cannot be obtained merely by implementing 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. We will demonstrate in the rest of the paper our effort toward an object-oriented approach to coping with software design faults.

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 [5]. 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 and software fault-tolerance schemes

An *idealized component* is a well-defined component that includes both normal and abnormal responses in the interface between interacting components, in a framework which could minimize the impact on system complexity [1, 5]. The above part of Figure 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 a limited form of software fault tolerance; 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 that is robust in the sense that catastrophic failure can be averted.

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 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 [2], a newer one being $t/(n-1)$-variant programming [11].

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. A well-known example of the use of dynamic redundancy are recovery blocks [7].

2.2: 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 lower part of Figure 1 suggests an abstraction model (or architecture) of a fault-tolerant software component and shows the details of the *controller* component which contains multiple subcomponents: *redundant variants of diverse design* and *an adjudicator* [1]. 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.

Our concept of an *idealized component with diverse design* is a natural extension and generalization 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 uniform 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.

The whole organization is in principle fully recursive. Each of the variants, the adjudicator and the controller 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 Figure 1, in a variant that is a subcomponent of the controller, three classes of exceptional situation are distinguished: interface exceptions, local exceptions, and failure exceptions.

Although the system structuring discussed above reflects a traditional functional view of software design, the abstraction architecture is equally appropriate for object-oriented programming. The object-oriented paradigm fits closely with the idea of idealized components. An ideal component can conveniently be thought of as an object [5]. 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.

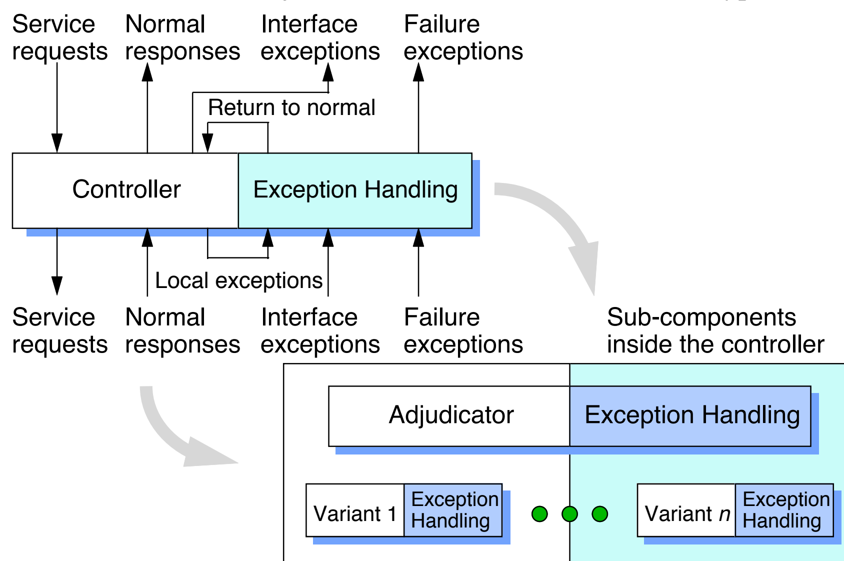

Figure 1 Idealized component with diverse design.

Based on the concept of abstract data type, it is natural to describe the 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. 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.

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 part of an operation, (2) different objects, and (3) different classes (that is, 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 operation.

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.

Class-level: Redundancy at class-level is usually considered 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 [3]. 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.

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 that 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 that 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 (that is, diverse design is limited to the operation-level); (2) as special objects that 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 — each variant is executed in parallel whenever invoked.

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 that 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, that is, 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 that corresponds to those of the other up-to-date variants. This can easily be done if the variants have the same internal data structure. 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.

In practical design of a fault-tolerant software system, we have to make an appropriate tradeoff between the two conflicting 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: predefined classes for software redundancy

The abstraction architecture discussed earlier 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. We will now briefly describe a possible object-oriented implementation by introducing several special classes — more technical program details can be found in [8]. We use a (C++)-like notation to demonstrate how software fault tolerance could be provided in mainstream languages, though our classes can also be simply described in other languages. Implementation issues will be further discussed in the fourth section, where other possible implementations are evaluated.

We first introduce the following `variant` class, which characterizes a variant component:

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

The function of the predefined `variant` class is to provide an interface through which the application programmer can develop particular concrete program variants; furthermore, the controller can organize the execution of these user-defined variants so as to provide either static or dynamic redundancy. Class `variant` can be viewed here as an abstract base class. A dummy definition is provided for the `variantDefinition()` function that is to be overridden in the derived (and user-defined) classes. The abstract base class may provide a set of standard exception handlers (for example, address out of range, divide by zero, invalid operation code) which deal with some local errors detected during the execution of those user-defined variants.

A basic set of adjudication algorithms (such as 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. (The `adjudicator` class, again as an abstract base class, is omitted here due to the limitation of space.)

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, pv2, ...;   //pointers to variants
public:
    sftStatus recoveryBlock(pa, pv[ ], n,...);   //execution modes
    sftStatus nVersionProgramming(...);
    sftStatus sequentialExecution(...);
    sftStatus dynamicExecution(...);
    ... ...
    void exceptionHandler1(...);
    ... ...
};
```

Here, class `controller` involves a set of control modes corresponding to different software fault-tolerance schemes. Other modes are possible and, by inheritance, they can be defined and implemented by the user as appropriate.

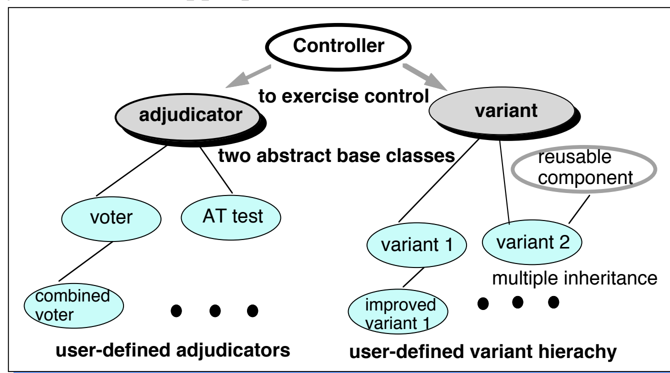

Figure 2 Predefined classes and user-defined class hierarchy.

To summarize, Figure 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

architecture, at least notionally, very generally applicable. 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):

```
sftModule()
{ ... ...
    controller* pf;
    acceptanceTest* pa;           //user-defined AT test
    variant1* pv[1] = &primary;   //user-defined primary
    variant2* pv[2] = &alternate1; //user-defined alternates
    ... ...
    status = pf->recoveryBlock(pa, pv[ ],...);
    ... ...
};
```

If the programmer wishes to use NVP 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. This makes it possible to write fault-tolerant programs in a terse, but disciplined style, thereby benefiting reliability. In the style of object orientation, the application programmer can construct an object-oriented system in such a way that 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, such as `sort()`, the operation which may contain the above code would cause the invocation of a group of diverse operations, for example, `sortVariant1()` or `sortVariant2()`, 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 predefined 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, thus 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 that 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 of adherence to these guidelines 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 a 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 preprocessor 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 in some circumstances during program development have to work in terms of the C++ program generated by the preprocessor, rather than the extended C++ program they have written.

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. But the latter has to be the responsibility of the application programmers. Special language features and/or programming conventions therefore cannot be avoided completely. The key problem would be twofold: how a set of simple (thus easy to check) language features can be developed with powerful expressibility, and how the supporting mechanisms such as those for state restoration can be provided in a 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 that 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 and an extension of the semantics of operation calls to the invocation of an object group. The Arche language [3] declares the object group as a sequence via the type constructor `SEQ OF` and provides the multioperation facility that supports operation calls on an object group. This new language can simplify the expression of some scheme like *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 uniform framework because the semantics of multioperations is statically fixed. What we desire is that the extended semantics of operation calls contain different control modes that would correspond to sequential, adaptive, or concurrent invocation of the object variants.

4.2: Reflection and reflective architecture

A reflective system can reason about, and manipulate, a representation of its own behavior [6]. This representation is called the system's meta-level. Reflection improves the effectiveness of the object-level (or base-level) computation and provides powerful expressibility by dynamically modifying the internal organization (the meta-level representation) of the system. In a reflective 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 program is kept simple and elegant.

The abstraction architecture described earlier helps the separation of object-level and meta-level descriptions. At object-level (or base-level), just linguistic supports for object groups and multiobject operations would be required. The controllers that 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 (such as 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 multilevel modularity and facilitates dynamic change in the system behavior. Figure 3 illustrates our 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 `robustData`. The `robustData` type provides an operation `int getMin()` that returns the smallest integer in the set of data. A call to `getMin()` on `G` would have the form `m = G.getMin()`. This operation call would then be reified at meta-level following the appropriate control mode, such as the recovery block mode. That is, the `getMin()` operation would be applied on the objects in `G` sequentially, depending upon error detection. This mode is further reified at the higher meta-level to involve automatic state restoration upon error 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 to be associated with the base-level code or to incorporate new meta-code into meta-levels whenever it is necessary for certain applications.

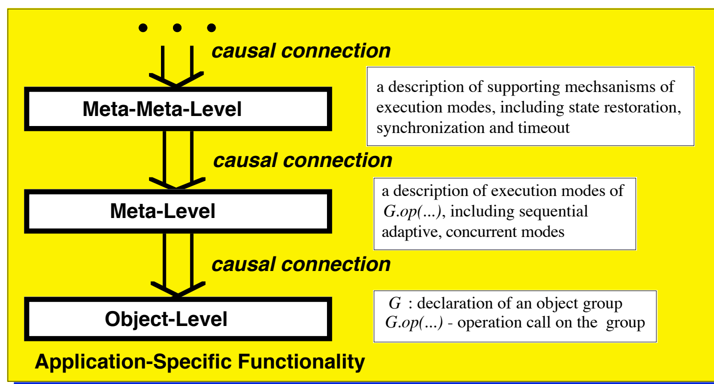

Figure 3 Reflective architecture for software fault tolerance.

In practice, we need a language that is both popular and reflective. The research in [4] describes an extension of C++ to provide a limited form of computational reflection, called Open C++. Based on this language our reflective architecture for software redundancy is undergoing evaluation and experimentation. 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 fault tolerance in an object-oriented fashion. A set of predefined classes demonstrates the usefulness of the proposed architecture. Depending on the actual language chosen, use of the architecture may depend to some degree on the programmer's adhering to a set of programming conventions. The meta-level technique is particularly useful in making our architecture reflective. As shown in the latter sections of the paper, such actions as object creation/destruction and operation invocation could be temporarily and transparently enhanced. Thus what would otherwise have depended on the programmer's adhering to particular conventions could instead be automatically invoked on his behalf. Such power of reflective capabilities, however, needs further verification and evaluation, especially in complex concurrent systems.

Acknowledgments

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

References

[1] T. Anderson (Ed.). *Resilient Computing Systems*, Collins Professional and Technical Books, 1985.

[2] A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault Tolerance During Program Execution," in *COMPSAC'77*, pp.149-155, Chicago, 1977.

[3] M. Benveniste and V. Issarny, "Concurrent Programming Notations in the Object-Oriented Language Arche," Res. Report, no.1822, Rennes, France, INRIA, 1992.

[4] S. Chiba and T. Masuda, "Designing an Extensible Distributed Language with a Meta-Level Architecture," in *ECOOP'93*, pp.482-501, Germany, 1993.

[5] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*, Second Edition, Prentice-Hall, 1990.

[6] P. Maes, "Concepts and Experiments in Computational Reflection," in *OOPSLA'87*, *ACM SIGPLAN Notices*, vol.22, no.12, pp.147-155, 1987.

[7] B. Randell, "System Structure for Software Fault Tolerance," *IEEE TSE*, vol.1, no.2, pp.220-232, 1975.

[8] B. Randell and J. Xu, "Object-Oriented Software Fault Tolerance: Framework, Reuse and Design Diversity," *PDCS2 1st Year Report*, vol.1, pp.165-184, Toulouse, 1993.

[9] L. Strigini, "Software Fault-Tolerance," *PDCS1 1st Year Report* , vol.2, Newcastle, 1990.

[10] U. Voges, ed. *Application of Design Diversity in Computerized Control Systems*, Springer Verlag, 1988.

[11] J. Xu, "The t/(n−1)-Diagnosability and its Application to Fault Tolerance," in *FTCS-21*, pp.496-503, Montreal, 1991.