Implementing Software-Fault Tolerance in C++ and Open C++: An Object-Oriented and Reflective Approach

Jie Xu, Brian Randell and Avelino F. Zorzo

Department of Computing Science
University of Newcastle upon Tyne, Newcastle upon Tyne, UK

Abstract

This paper reports our experience with the use of the C++ language and Open C++ (a reflective version of C++) to implement reusable, dependable control structures that support the prevision of software-fault tolerance in the application layer. We first implement the support using an object library approach and then re-design it using a reflective one. We demonstrate through a realistic experiment why reflection and metaobject protocols are particularly suitable for the development of fault-tolerant programs.

Key Words — C++ and Open C++, linguistic mechanisms, metalevel interface, metaobject protocols, object-oriented programming, reflective system architecture, software-fault tolerance.

1: Introduction

A promising approach to the improvement of software dependability is the application of some new ideas in program structuring, and particularly *object-oriented programming*, to the control of system complexity. However, despite of various new techniques, a realistic software system will still contain residual software design faults given the complexity of today's computing systems [8]. Approaches and mechanisms are therefore required to enable a system to tolerate software faults remaining in the system after its development. Practical techniques do exist and have been proved successful. (The latest Wiley book [9] presented a comprehensive and detailed survey of software-fault tolerance issues, where further references can be found.)

In this paper we introduce our experience with the use of the C++ language [14] and Open C++ [3] (a reflective version of C++) to implement *reusable*, dependable control structures that can effectively support the prevision of software-fault tolerance in the application layer of object-oriented software. We emphasize the careful separation of obligations and concerns which reduces the burden of a specific programmer (e.g. user of a class, designer of a fault-tolerant program, and system programmer) and thus decreases the probability of producing software faults.

In order to compare the usual implementation approaches (e.g. the object-library approach) with the reflective one [10] based on metaobject protocols [6], we implement a set of control mechanisms for software-fault tolerance in C++ and Open C++ respectively. The reflective meta-level interface hides most implementation details of a fault-tolerant mechanism from the application-level and enables the programmer to flexibly change the use of different fault tolerance schemes without having to change the code of application programs. Experiment-based analysis in this paper verifies the effectiveness of both approaches and shows acceptable runtime overheads.

The remainder of this paper is organized as follows. In Section 2 we describe our model and an implementation framework for software-fault tolerance and address the separation of concerns with respect to the development of fault-tolerant programs. Section 3 demonstrates how to use C++ to define and implement the object library that supports software-fault tolerance. Section 4 introduces the reflective implementation in Open C++ and the related metaobject protocols. We describe our experiment setting in the fifth section and provide testing results and the performance-related analysis.

2: Software-Fault Tolerance: Schemes and an Implementation Framework

Software-fault tolerance, in the context of this paper, is concerned with all the techniques necessary to enable a system to tolerate software design faults. 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 [8]. 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.

Because software faults are permanent in nature, techniques for software-fault tolerance in principle require redundancy of design and/or of data and/or of environment (e.g. different processors). Design redundancy (or design

diversity) is the approach in which the production of two or more components is aimed at delivering the same service through independent designs and realizations [1][7]. The components, produced through the design diversity approach, or more generally involving data diversity and environment diversity, from a common service specification, are called *variants*. By incorporating at least two variants of a system, tolerance to design faults necessitates an *adjudicator* (or a decision algorithm) that provides an (assumed to be) error-free result from the execution of variants. To coordinate the execution of variants and the final adjudication, a *control mechanism* or controller is required. (For more details of a general framework for software redundancy, see Figure 1 and [17].) Classical techniques for tolerating software design faults are mostly based on some form of design diversity, including recovery blocks (RB) [11] and *N*-version programming (NVP) [1].

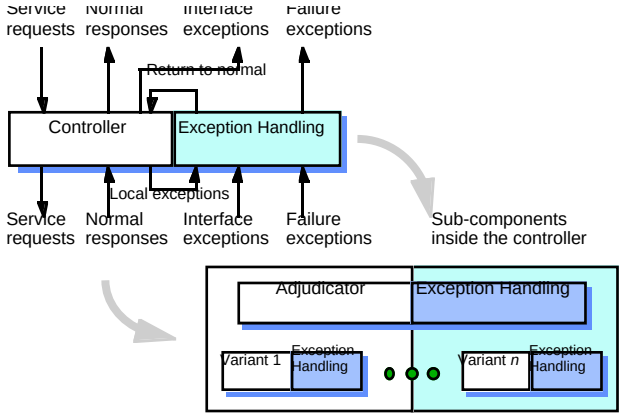

Figure 1 A fault-tolerant component with diverse designs; reproduced from [17].

The first scheme designed to provide software-fault tolerance was the recovery block scheme (RB). In this approach, variants are named alternates and the main part of the adjudicator is an acceptance test that is applied sequentially to the results produced by variants: if the first variant fails to pass the test, the state of the system is restored and the second variant invoked on the same input data and so on, sequentially, until either some result passes the acceptance test or all the variants are exhausted. Software variants are organized in RB in a manner similar to the standby sparing techniques (dynamic redundancy) used in hardware and may be executed serially on a single processor. *N*-version programming (NVP) is a direct application of the hardware *N*-modular redundancy approach (NMR) to software. A voting mechanism determines a single adjudication result from a set or a subset of all the results of variants which are usually executed in parallel. Apart from the two well-known approaches, a number of other schemes for software-fault tolerance have been developed such as *N* Self-Checking Programming [7], *t*(*n*-1)-Variant Programming [18], and Self-Configuring Optimal Programming [15] for sequential programs, and Conversations [11] and Coordinated Atomic Actions [16] for concurrent systems. Due to the limitation of space we will not deal with them further.

Complexity control is particularly crucial in designing and using a fault-tolerant software system. In notion, we classify programmers into three classes with respect to a fault-tolerant object-oriented program: the **users** of fault-tolerant objects, the designers of fault-tolerant objects (or **ft-designers**) and the **system** (or meta-level) **programmers**. If just using a fault-tolerant object or an improved version of a original object, a user needs to know a little information about whether the interested object is fault-tolerant or not. However an ft-designer is required to produce a fault-tolerant program and thus needs to know more about the schemes for coping with software-faults and is responsible for developing variants and the related adjudicator. He/she also has to choose a special scheme, such as RB or NVP, for the development of the fault-tolerant program. Note that the control mechanism for a fault tolerance scheme should be application-independent and so can be made reusable. It is the system programmer who is responsible for providing the ft-designer with the high-level programming interface for various software fault tolerance schemes, which hides the implementation details of their control mechanisms. Of course, what we emphasize here is the separation of different concerns; in practice a person may well play several different roles.

Finally, a supporting system for the development of fault-tolerant programs may be defined in a very abstract form. We need however a more detailed description of the target system in order to examine implementation-related issues. Figure 2 shows a typical supporting system in which objects may be located in different nodes connected by a communication network and provide services collectively by passing messages among them.

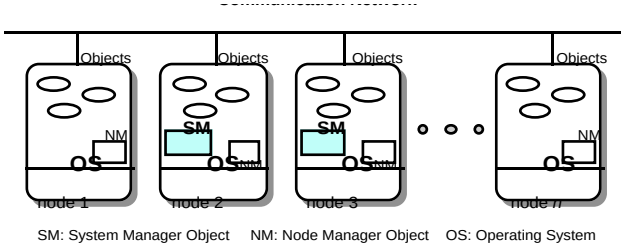

Figure 2 A distributed supporting system.

3: C++ and the Object Library Approach

C++ [14] is an object-oriented superset of the well-known C language, and includes new facilities for type-inheritance, data abstraction, and operator overloading. Abstraction and inheritance are based on the concept of a class: instances of a class are objects with specific operations provided for their manipulation. Furthermore, given a base class, a sub-class of the base class can be

defined so as to inherit some or all of the attributes of the base class.

There are several possible solutions to the prevision of supports for software-fault tolerance in the application layer [17], but most of previous proposals and implementations were based on pre-defined classes and run-time libraries. Huang and Kintala of AT&T Bell Labs. [5] developed three software reusable components in C that provide software-fault tolerance in the application layer, supporting fault-tolerant structures like checkpointing and recovery, replication, recovery blocks, *N*-version programming, exception handling, re-try blocks etc. Their modules have been ported to a number of UNIX platforms, already applied to some new telecommunications products in AT&T and the performance overhead due to these components has been shown to be acceptable.

The research group at Newcastle has recently developed a set of reusable components in C++, providing high-level object-oriented programming interfaces for software-fault tolerance [12][17]. These components support both forward and backward error recovery, recovery blocks, *N*-version programming, self-configuring optimal programming etc. All details of control mechanisms are hidden from the ft-designer who can simply choose different fault-tolerant structures by employing inheritance and polymorphism mechanisms. Real programs examples are also tested and presented.

To clearly explain the library approach, let us first consider a simple example of a sorting application. This example is made up of an application program (as the client) and three sorting servers. Each sorting server implements a variant of the sorting operation by receiving an object which contains a list of integer numbers, sorting the list, and then sending the result back to the application.

Given a base class called *ArrayList*, it is associated with a set of operations (e.g. *Sort*) which can be re-defined and implemented in an alternative, more dependable manner, by means of the inheritance mechanism. When creating an array object, the user of the class (or object) should specify the required type of *ArrayList* such as `NORMAL, QUICK, HEAP, SHELL, RB, NVP` etc. The constructor of class *ArrayList* will create a corresponding object as to different *kind* parameters. In C++ the *Sort* operation may be implemented as virtual in the base class. It can be redefined later in the sub-classes of *ArrayList*.

```
class ArrayList {
    public:
           ArrayList(int kind);
           virtual int Sort();
           //other operations ...
    protected:
           Vector list(MAXELEM);
           int size;
};
```

Assume that there exist some C++ library programs provided by the system programmers that implement the control mechanisms for different fault tolerance schemes. The *FTClass* class below provides several interfaces for the control of recovery blocks, *N*-version programming, and other schemes.

```
template <class TF> class FTClass {
  public:
    FTClass();
    void Variant(int port, char *machine);
    bool RecoveryBlock(TF& obj);
    bool NVersionProgramming(TF& obj);
    // other control structures
  private:
    // some internal states
};
```

In this class definition, `TF` is the class/type of the object that will be manipulated by the variants in the remote servers. The *port/machine* parameters of the *Variant()* operation are the address of the computer that contains the server for one of the variants used by *RecoveryBlock* or *NVersionProgramming*. Both control structures return a bool result indicating if the variants achieve an acceptable result (`TRUE`) or an incorrect result (`FALSE`). Since an adjudicator is often application-specific, the object passed to the fault-tolerant control structures must provide a pre-defined *Adjudicator* operation. As an example, the control mechanism for recovery blocks could be implemented by the system programmer as follows.

```
template <class TF>
bool FTClass<TF>::RecoveryBlock(TF& obj)
{
 Socket<TF> s;
 TF bk = obj;
 for (int i=0; i<nvariants; i++) {
   // Try alternate/variant
    s.ConnectWrite(port[i],machine[i],obj);
    s.ReadClose(obj);
   if (!obj.Adjudicator()) {
    // Ensure test was not successful
    // recover the object and try
    // another alternate/variant
          obj = bk;
   }
   else
          return(TRUE);
   }
   // All alternates failed
   return (FALSE);
}
```

More precisely, the *RecoveryBlock* control structure receives an object and passes this object onto the first variant. When the result is returned, the adjudication operation is executed. If the result is acceptable then `TRUE` is returned, or otherwise the object (state) is recovered and the next alternate is executed. If all alternates fail then the `RecoveryBlock` operation returns an error signal (`FALSE`) indicating that an acceptable result cannot be obtained. (Note that sockets have been used in our particular

experiment. This can be easily changed by using a different communication protocol, such as those RPC mechanisms used in the Arjuna system [13].)

Now, by using the reusable control structures provided above, an ft-designer can easily construct dependable versions (or sub-classes) of the *ArrayList* class. The ft-designer has to develop the variants and adjudicator, and to indicate which type of objects will be handled by the *FTClass*.

```
int RBArrayList::Sort()
{
    FTClass<Vector> ftObj;
    bool error;
    ftObj.Variant(p1,"node1.glororan");
    ftObj.Variant(p2,"node2.bowes");
    ftObj.Variant(p3,"node3.yardhope");
    error = ftObj.RecoveryBlock(list);
    return error;
}
```

Advantages of this object library approach include i) no modification to the compiler so enabling rapid experimentations, and ii) some reduction of the extra burden (of fault tolerance design) on the ft-designers. However, the application programmer who wishes to utilize the library routines must be fully responsible for correct use of them; the programming conventions related to the use of library programs must be strictly adhered and all checks as to the adherence can be performed only by the programmer him/herself. Moreover, such conventions cannot make a clear separation between application code and the extra code for software-fault tolerance. This is of course a fruitful source of software faults.

4: Open C++ and the Reflective Approach

It is very important to notice a fundamental difference between simple replication and diversity in design/data/environment; the former could be made transparent to the programmer and performed automatically by a supporting system, but the latter generally requires the direct effort from the ft-designer. Simple (thus easy to check) language features with powerful expressibility will be particularly helpful in properly specifying software variants and the adjudicator.

A reflective system can reason about, and manipulate, a representation of its own behavior [10]. 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 in Figure 1 helps the separation of base-level and meta-level descriptions. The controllers that control the execution of variants and the adjudication are naturally implemented as meta-objects. The actual execution of an operation call is controlled and dynamically reified at meta-level. Figure 3 illustrates our reflective architecture in a distributed environment.

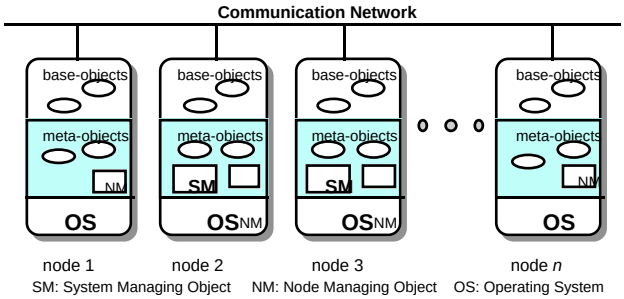

Figure 3 Reflective architecture in a distributed environment

Open C++ [3] is a reflective version of C++ that provides the programmer with two levels of abstraction: the base-level, like traditional C++ object-oriented programming; and the meta-level which allows certain aspects (e.g. semantics of the operation calls) of C++ to be redefined. In Open C++ operation calls to base-level objects can be *intercepted* at the meta-level by metaobjects.

We now introduce a reflective approach to the implementation of software-fault tolerance in the application layer, in which i) software variants and the adjudicator are written in C++ and implemented at the base level, together with a simple syntax extension that combines the variants and the adjudicator into a single place; ii) various control structures for different software-fault tolerance schemes are implemented in Open C++ at the meta level. Note that different schemes for software-fault tolerance can be selected dynamically (even at run time), with respect to special application requirements; and such dynamic changes are made through a run-time meta-level interface without having to make changes at the base level. With our approach, the additional burden on the ft-designer is further reduced and the code needed to support software-fault tolerance is separated clearly from normal programming at the base level. For example, the *FTMetaClass* class below provides the ft-designers with a clean interface for the use of various control structures for software-fault tolerance.

Suppose that one would like to make the *Sort()* operation fault-tolerant. We are able to first make the *Sort()* operation reflective and then extend its semantics by means of the directive "*//MOP reflect:*" of Open C++. Note that the code of the control mechanisms implemented

in *FTMetaClass* will not, at least in notion, appear in the source code of *ArrayList*. So the base-level programs will be very similar to normal, non-fault-tolerant ones.

```
template <class TM> class FTMetaClass :
public MetaObj {
 public :
  FTMetaClass(VariantList& list);
  void Meta_MethodCall(Id mid, Id cat,
        ArgPac& args, ArgPac& rep, TM& obj);
 private:
  VariantList vList;
};

template <class TM> void FTMetaClass<TM> ::
  Meta_MethodCall(Id mid, Id cat, ArgPac& args,
                    ArgPac& rep, TM& obj) {
 // Similar code used in
RBArrayList::Sort(),
 // or NVPArrayList::Sort()
       if (!obj.Adjudicator)
       Meta_HandleMethodCall(mid, args, rep);
}
```

5: Experimental Evaluation

In our experiment we use GNU C++ version 2.6.3 and Open C++ version 1.2 [2] as basic programming languages. The target environment consists of a set of workstations running UNIX and connected through TCP/IP (see Figure 4).

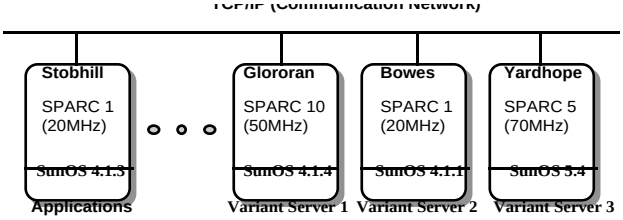

Figure 4 Target environment (hardware/software).

This experimental distributed system is organized in a client-server manner. Clients and servers communicate each other by means of TCP/IP BSD sockets. We believe that this communication facility is effective enough to support our experiment with respect to fault-tolerant computing and object-oriented development (since communication is not our major concern here, it is assumed that the used communication mechanisms are highly dependable). Other communication mechanisms may be considered in our further research, such as remote procedure calls (RPC) [13]. We define a special object that provides the functionality of communication through sockets, named *Socket*. Two operations are attached to *Socket*: *ConnectWrite()* that is responsible for establishing the connection between the client and the server and for sending an object to the server; and *ReadClose()* responsible for receiving the result (or object) from the server and for closing the connection.

A large sorting application is implemented following four variants of the basic sorting algorithm. In order to examine the behaviour and effectiveness of each implementation testing based on software-fault injection is performed. The execution time was measured in microseconds (μseconds) while the time of network communication is assumed as constant. A data set of 200 elements was used to measure the times.

Variant \ Test		1	2	3	4
Quick	Result	OK	fail	fail	fail
	Time	3198	397	397	3319
Heap	Result	OK	OK	fail	OK
	Time	12037	12037	9688	12037
Shell	Result	OK	OK	OK	OK
	Time	2788	2788	2788	2788
RB	Result	OK	OK	OK	fail
	Time	5403	15464	19574	5669
NVP	Result	OK	OK	fail	OK
	Time	17163	17163	14379	17163

Table 1 Performance-related testing results.

In Test One, every *Sort* operation was executed without fault injection, and all lists received were correctly sorted. It is interesting to notice that the dependable versions based on RB and NVP have longer execution time than the faster versions of the normal sorting operation. That run-time overhead is mainly due to the coordination of variants and the execution of the adjudicator. In addition, since NVP needs to wait for the completion of the slowest variant (e.g. *HeapSort* in the example), it has longer execution time than RB which uses *QuickSort* as its first alternate. However, due to the use of an acceptance test, the overhead of RB is slightly higher than that of the *QuickSort* operation itself.

In Test Two, several software faults were injected into the *QuickSort* server, by randomly commenting a line on the algorithm. Once this operation fails, the *RBArrayList::Sort()* will call its second alternate. Note that, compared with the results in Test One, the overhead of NVP is still the same, but the time of RB has increased because of the execution of the second variant.

In Test Three, design faults were also inserted into the *HeapSort* server; this causes the failure of three algorithms (i.e. Quick, Heap and NVP). The other two (Shell and RB) can still deliver correct computation. NVP fails because the voter failed to find the majority of three different results. This time, we can see that the time of NVP is shorter than that of RB since RB has to activate all its alternate to reach a correct result. The total overhead of RB is the execution time of all the variants plus the cost of state restoration and adjudication.

In the fourth test, some software faults which cannot be detected by the acceptance test were inserted into the *QuickSort* server, and both Quick and RB produce erroneous lists without signalling any error. The general overhead is similar to those in the first test. Note that NVP can tolerate such a fault very effectively.

Testing is conducted as well in our reflective implementation, similar to those summarized in Table 1. Tabel 2 shows the different run-time overheads between calling a C++ operation and calling an Open C++ (reflective) operation. The ratio indicates the extra cost caused by a reflective operation call. However, while combining the results of Table 2 with Table 1, we find this cost is only a very small part of the whole overhead imposed by fault tolerance mechanisms. (It will contribute a smaller part if the communication cost is taken into account.) More information about the overhead of reflective operation calls can be found in [3][4].

	Stobhill	Yardhope	Bowes	Glororan
Normal	56	3	64	8
Reflective	100	6	100	16
Ratio	1.78	2	1.56	2

Table 2 Time consumed by reflective operation calls.

6: Conclusions

There have been a few papers about experimental evaluation of software-fault tolerance schemes in the context of object-oriented programming. We have conducted an initial experiment using two different implementation approaches: the C++ object library and the metaobject protocol in Open C++. Our programming experiment shows that the run-time overhead of software-fault tolerance is generally acceptable while making a clear, structured separation of concerns in both design and operation stages. The metaobject approach provides an ft-designer with the clearer and simpler interface but with a slightly higher run-time overhead. However, when the communication cost is considered, the overhead imposed by reflective operation calls will not be of major concern.

Acknowledgements

This work was supported by the CEC-sponsored ESPRIT Basic Research Actions 3092 and 6362 on Predictably Dependable Computing Systems, and by the ESPRIT Long Term Research Project 20072 on Design for Validation (DeVa). Avelino Zorzo is also supported by CNPq (Brazil) under grant n. 200531/95.6.

References

[1] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Trans. Soft. Eng.*, vol. SE-11, no.12, pp.1491-1501, 1985.

[2] S. Chiba. *OpenC++ Programmer's Guide*, Technical Report 93-3, Dept of Information Science, Univ. of Tokyo, Japan, 1993.

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

[4] J. Fabre, V. Nicomette, T. Perennou, R.J. Stroud and Z. Wu, "Implementing Fault Tolerant Application Using Reflective Object-Oriented Programming," In *Proc. 25th Int. Symp. on Fault-Tolerant Computing*, Pasadena, pp.489-598, June 1995.

[5] Y. Huang and C. Kintala, "Software Implemented Fault Tolerance: Technologies and Experience," in *Proc. 23th Int. Symp. Fault-Tolerant Computing*, Toulouse, pp.2-9, June 1993. (An expanded version of this paper is: "Software Fault Tolerance in the Application Layer," in *Software Fault Tolerance*, ed. M. Lyu, Trends in Software series, WILEY, 1995, pp.231-248.

[6] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol*. MIT Press, 1991.

[7] J.C. Laprie, J. Arlat, C. Beounes, K. Kanoun and C. Hourtolle, "Hardware and software fault tolerance: definition and analysis of architectural solutions," in *Proc. 17th Int. Symp. Fault-Tolerant Computing*, Pittsburgh, pp. 116-121, June 1987.

[8] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*, Second Edition, Springer-Verlag, 1990.

[9] M. Lyu (ed.),*Software Fault Tolerance*, Trends in Software series, WILEY, 1995.

[10] P. Maes, "Concepts and Experiments in Computational Reflection," in *Proc. OOPSLA 87*, pp.147-155, 1987.

[11] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Soft. Eng.*, vol. SE-1, no.2, pp.220-232, 1975.

[12] C.M.F. Rubira-Calsavara and R.J. Stroud, "Forward and Backward Error Recovery in C++," *Object-Oriented Systems*, vol.1, no.1, pp.61-85, 1994.

[13] S.K. Shrivastava, G.N. Dixon and G.D. Parrington, "An Overview of the Arjuna Distributed Programming System," *IEEE Software*, vol.8, no.1, pp.66-73, 1991.

[14] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, 2nd Edition, 1991.

[15] J. Xu, A. Bondavalli and F. DiGiondomenico, "Dynamic Adjustment of Dependability and Efficiency in Fault-Tolerant Software," in *Predictably Dependable Computing Systems* (eds. B. Randell et al.), the *ESPRIT Basic Research* series, Springer-Verlag, pp.155-172, June 1995.

[16] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu, "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery," In *Proc. 25th Int. Symp. on Fault-Tolerant Computing*, pp.499-508, Pasadena, June 1995.

[17] J. Xu, B. Randell, C.M.F. Rubira-Calsavara and R.J. Stroud, "Toward an Object-Oriented Approach to Software Fault Tolerance," in *Fault-Tolerant Parallel and Distributed Systems*, IEEE CS Press, pp.226-233, Sept. 1995.

[18] J. Xu and B. Randell, "Software Fault Tolerance: $t/(n-1)$-Variant Programming," *IEEE Trans. Reliability*, vol.45, no.2, June 1996.