University of Newcastle upon Tyne

COMPUTING LABORATORY

A Cost-Effective and Flexible Scheme for Software Fault Tolerance

A. Bondavalli, F. Di Giandomenico and J. Xu

TECHNICAL REPORT SERIES
No 372 February, 1992

TECHNICAL REPORT SERIES
No. 372 February, 1992

A Cost-Effective and Flexible Scheme for Software Fault Tolerance

A. Bondavalli, F. Di Giandomenico and J. Xu

Abstract

A new software fault tolerance scheme, called the Self-Configuring Optimistic Programming scheme, (SCOP), is proposed. It attempts to reduce the cost of fault tolerant software by providing designers with a flexible redundant system component by which reliability and effectiveness can be dynamically combined. SCOP is based on comparisons of results produced by the variants, delivers results which are judged correct with the required probability, and is structured in phases in order to release such results as soon as available. It can be parameterized with respect to both the desired reliability and the desired response time. SCOP thus allows a trade-off between various attributes of system services (such as reliability, throughput and response time) in a desired manner. In offering this choice, SCOP also tries to minimize resource usage and is thus a flexible and cost-effective tool for gracefully degradable systems.

Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

BONDAVALLI, Andrea

A Cost-Effective and Flexible Scheme for Software Fault Tolerance
[By] A. Bondavalli, F. Di Giandomenico and J. Xu

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Laboratory, 1992.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 372)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series. 372
DI GIANDOMENICO, Felicita
XU, Jie

Abstract

A new software fault tolerance scheme, called the Self-Configuring Optimistic Programming scheme, (SCOP), is proposed. It attempts to reduce the cost of fault tolerant software by providing designers with a flexible redundant system component by which reliability and effectiveness can be dynamically combined. SCOP is based on comparisons of results produced by the variants, delivers results which are judged correct with the required probability, and is structured in phases in order to release such results as soon as available. It can be parameterized with respect to both the desired reliability and the desired response time. SCOP thus allows a trade-off between various attributes of system services (such as reliability, throughput and response time) in a desired manner. In offering this choice, SCOP also tries to minimize resource usage and is thus a flexible and cost-effective tool for gracefully degradable systems.

About the author

Andrea Bondavalli joined CNUCE of CNR in 1986 where he has been 'Ricercatore' since 1988.
Felicita Di Giandomenico joined IEI of CNR as 'Ricercatrice' in 1989.
Jie Xu is currently a Research Associate in the Computing Laboratory which he joined in April 1990.

Suggested keywords

ADJUDICATION MECHANISMS
COST-EFFECTIVENESS
FAULT-TOLERANT ARCHITECTURES
RELIABILITY ASSESSMENT
SOFTWARE FAULT TOLERANCE

Suggested classmarks (primary classmark underlined)

Dewey (18th):	001.64404	621.381958
U.D.C.	519.687	681.322.02

A Cost-Effective and Flexible Scheme for Software Fault Tolerance

Andrea Bondavalli*, Felicita Di Giandomenico** and Jie Xu***

*CNUCE-CNR, Pisa, Italy;
**IEI-CNR, Pisa, Italy;
***Computing Laboratory, University of Newcastle upon Tyne

Abstract

A new software fault tolerance scheme, called the Self-Configuring Optimistic Programming scheme, (SCOP), is proposed. It attempts to reduce the cost of fault tolerant software by providing designers with a flexible redundant system component by which reliability and effectiveness can be dynamically combined. SCOP is based on comparisons of results produced by the variants, delivers results which are judged correct with the required probability, and is structured in phases in order to release such results as soon as available. It can be parameterized with respect to both the desired reliability and the desired response time. SCOP thus allows a trade-off between various attributes of system services (such as reliability, throughput and response time) in a desired manner. In offering this choice, SCOP also tries to minimize resource usage and is thus a flexible and cost-effective tool for gracefully degradable systems.

Keywords: Software Fault Tolerance, Fault-Tolerant Architectures, Reliability Assessment, Adjudication Mechanisms, Cost-Effectiveness.

I. Introduction

Computing system *dependability* refers to the quality of the delivered service such that reliance can be justifiably placed on this service, and serves as a generic concept encompassing notions of reliability, maintainability, availability, safety, functionality, performance, timeliness, etc. [1, 14]. A dependable computing system is capable of providing *dependable service* to its users over a wide range of potentially adverse circumstances. The development of dependable computing systems consists in the combined utilization of a wide range of techniques, including *fault tolerance* techniques intended to cope with the effects of faults and avert the occurrence of

---

This work has been partially supported by the Commission of the European Communities in the framework of the Esprit BRA project 3092 PDCS. Andrea Bondavalli and Felicita Di Giandomenico are currently in sabbatical at the Computing Laboratory of the University of Newcastle upon Tyne.

failures or at least to warn a user that errors have been introduced into the state of the system [1]. As we know, the provision of means of tolerating anticipated hardware faults has been a common practice for many years. A relatively new development concerns the techniques for tolerating unanticipated faults such as design faults (typically software faults). Software fault tolerance is always based on the principle of *design diversity*. Design diversity may be defined as the production of two or more systems (e.g., software modules) aimed at delivering the same service through independent designs and realizations [6]. The systems, produced through the design diversity approach from a common service specification, are called *variants*. Incorporating two or more variants of a system, tolerance to design faults necessitates an *adjudicator* [2], which is based on some previously defined decision strategy and is aimed at providing (what was assumed to be) an error-free result from the outcomes of variant execution. Several well documented techniques for tolerating software design faults are recovery blocks (RB)[20], N-version programming [4], N self-checking programming [16], t/(n-1)-Variant Programming [23] and the certification trail scheme [22], and some intermediate or combined techniques[11, 21].

The first scheme for achieving software fault tolerance to be developed was the recovery block scheme (RB). In this approach, the variants are named alternates and the main part of the adjudicator is an acceptance test that is applied sequentially to the results of the variants: if the first variant (primary alternate) fails to pass the acceptance 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. Most of the time, RB involves very low structural and operational time overheads unless faults occur. It is therefore highly efficient. On the other hand, the acceptance test is used to provide a last line of defence for detecting errors and, in general, is derived from the semantics of the specific applications. Sometimes, it may be difficult to identify a proper acceptance test, and the close design dependency between the acceptance test and alternates may have adverse impact on the reliability of the whole system. The three approaches discussed below are based on the parallel execution of multiple variants (although sequential execution is conceptually possible).

In the N-version programming approach (NVP), the adjudicator performs an adjudication function on the set of results provided by the variants. NVP is sensitive to the adjudication strategy used, as different adjudication functions can be utilized. Secondly, in N-self-checking programming (NSCP), fault tolerance is attained by the parallel execution of N self-checking software components. Each self-checking software component may be built either from one variant with an associated acceptance test or from the association of a pair of variants with a comparator (we will only address the latter throughout this paper). One of the components is regarded as the active component, and the others are considered as "hot" stand-by spares. Upon failure of the active component, service delivery is switched to a "hot" spare. The t/(n-1)-variant programming scheme (t/(n-1)-VP) is the third scheme which is developed based on system level diagnosis theory. Even though fault tolerance does not require diagnosis, automatic fault diagnosis can attain fault tolerance without performance degradation and furthermore can ease maintenance operations. This approach uses a particular diagnosability measure, t/(n-1)-diagnosability, and can isolate the faulty modules within a set of at most (n-1) variants. By

applying a t/(n-1)-diagnosis algorithm to some of the results obtained by the parallel execution of n variants, it selects a presumably correct result as the output. The adjudication functions of these three approaches are usually based on result comparison, rather than the use of application related Acceptance Test on a single result, so showing better characteristics of independence between the adjudicator and the variants. NVP, NSCP and t/(n-1)-VP have fixed response time and guarantee to give timely responses, but may consume excessive system resources unnecessarily. Some proposals, [18], tried to address this problem and to present some possible solutions.

The certification-trail scheme (CT) presents a novel technique for achieving software fault tolerance. The central idea of CT is to execute an algorithm so that it leaves behind a trail of data (certification trail) and, by using this data, to execute another algorithm for solving the same problem more quickly. Then, the outputs of the two executions are compared and are considered correct only if they agree, otherwise other algorithms are executed. This technique always requires time redundancy and may be plagued by data dependency.

From a global point of view, some qualitative observations and considerations can be made:

1) The above fault tolerance schemes are not defined directly according to the application needs concerning the quality of the results and the adjudged reliability of the hardware components or of the variants, but instead such information is used to provide strong fault hypotheses on the number and the nature of faults to be tolerated. Only after these strong fault hypotheses have been derived, is the architecture of the scheme designed and the degree of redundancy necessary for tolerating the worst occurrence of faults chosen.

2) Most of these schemes, other than RB, execute all of their variants regardless of the state of the system (normal or faulty). Moreover, some schemes, such as NVP and t/(n-1)-VP, are aimed at providing the 'best' possible result without regarding for whether the reliability obtained is higher than that strictly required, or if the required reliability could be achieved with less effort. This is because they always assume that the maximum number of faulty components may be present in the system; but since this worst case rarely happens, the amount of resources consumed is often higher than necessary. In this sense, they are not cost-effective.

3) A scheme can be classified as a *syntactic scheme*, if its adjudication function is based on the result comparison, or a *semantic scheme* when the adjudication is absolute, i.e., uses an absolute judgement on the single individual result based on the application semantics. As for error recovery, concurrency control, and many other topics, syntactic schemes have a wider applicability range than semantic ones. Furthermore, syntactic adjudication functions might have a lower probability of failure than semantic ones (e.g., acceptance tests).

In more detail, Table I summarizes some key characteristics of the above schemes for software fault tolerance.

Scheme	Variant execution	Time Overhead	Cost-Effectiveness	Adjudication Base	Manifest Dependency	Tolerance to Related Faults
RB	Sequential	Low if no faults	Good	Semantic	AT and Variants	Application Dependent
NVP	Parallel n > 2	Low	Poor	Syntactic	Variants	Relatively Good
NSCP	Parallel N=2n>1	Low	Poor	Syntactic	Variants	Relatively Poor
t/(n-1)-VP	Parallel n > 2	Low	Poor	Syntactic	Variants	Relatively Good
CT	Sequential	High	Acceptable	Syntactic	Data	Relatively Poor

Table I. Some key characteristics of software fault tolerance schemes

In this paper, we propose a new software fault tolerance scheme, called **Self-Configuring Optimistic Programming** (SCOP), which is aimed at improving the cost-effectiveness of fault-tolerant software (diminishing the waste of resources) by providing designers a flexible system component in which reliability and cost-effectiveness can be dynamically combined. SCOP is based on result comparisons, and is organized in dynamic phases in order to deliver a result immediately once that the result has been assessed to have the required probability of being correct.

The rest of the paper is organised as follows. Section II introduces some important concepts and definitions of software fault tolerance, especially those concerning the quality of the delivered service. The detailed description of the proposed scheme is given in the third section. In the fourth section we propose a new design methodology for building fault-tolerant software. Section V presents a reliability and efficiency evaluation and a comprehensive analysis of the main software fault tolerance schemes. We conclude this paper in Section VI.

II. Basic Concepts and Fault Assumptions

Given the existence of a complete, unambiguous specification we can define the *reliability* R(t) *of a system* as a function of time, expressing the conditional probability that the system will conform to its specification throughout the interval $[T_0, t]$, given that the system was performing correctly at time T_0 [10, 17]. The reliability of a system can be reflected indirectly on the quality of the delivered service, and thus users can justifiably place their reliance on this service. However, by and large, the quality of the delivered service will vary with respect to the different states of the system (normal or faulty). We have noticed that, for all fault tolerance approaches, the quality of a result that is output by a fault-free system will differ from that of a result output by the system when containing some faulty components or variants. This phenomenon is most obvious in NVP. For instance, a service, delivered by a 3VP system, based on three consistent results of the variants has a higher probability of being correct than one based on a simple majority of three results, i.e., two consistent and one different result. It is therefore necessary to define a new measure of the quality of a single result delivered by the system. We define the new measure below.

The *reliability of a result* R_r, delivered by a system, can be characterized by the probability that this result is correct, i.e., the probability that this result conforms to its specification, which may well concern the result's timing as well as its value.

The reliability of a result is independent of time: it will not change once that the result is generated, but it is usually different with respect to the different *syndromes* which indicate the distinct states of the system. A syndrome is a set of information (in general, involving those results produced by the variants) used by an adjudicator to perform its judgement as to the correctness of a result. Due to the changeable probabilities corresponding to the distinct syndrome, designers of fault tolerance schemes often have to sacrifice the cost-effectiveness of the system (e.g., by consuming excessive useful resources) so that the system can provide a service with the required (satisfactory) reliability when the rare worst case really occurs, although most of the time the reliability of a result may be much higher than that strictly required. From this angle, we expect to develop a new scheme in order to minimize the waste of resources for given reliability requirements.

We will now address three fundamental issues on the design of fault-tolerant software. First, when designing such a software system, we must determine the criterion for the system's delivering a result. For example, one can use the reliability requirement of users to derive a maximum number of faulty components admitted in the system. Then, different adjudication functions can be defined based on this fault assumption and the redundancy degree needed in the system can be decided (e.g., for a majority function, 2k+1 components may be required under the hypothesis of at most k faults). Another example is to assume a maximum number of components that may produce consistent, but incorrect results, and then develop an adjudication function under this assumption, e.g., the plurality voter [19]. The third possible criterion might be to release directly each single result according to the reliability requirement of users without assuming any upper bound on the number of admitted faults. The corresponding adjudication function simply releases the result that have its reliability equal to or greater than the required one. The most popular (syntactic) fault tolerance schemes in the literature adopt criteria like the first two described. The third method will possess advantages including a more precise management of the single services to be provided and a better calibration of the degree of reliability to be pursued, which can lead to higher cost-effectiveness. The main problem in adopting this method lies in getting sufficiently precise probabilities of failure of the software components [8].

The second issue concerns *flexibility* in the degree of reliability of services to be provided by a scheme. This kind of flexibility will be associated to the individual service, meaning that two different services may be required with different levels of reliability (at least) during the time period of system life. A flexible scheme must be ready to deliver services with different reliability when requested. There are of course different extents in designing such a scheme: one extreme is the scheme without any flexibility, and the other is the scheme that treats the reliability level as a parameter for any service. The former is meant to provide all services with the same degree of reliability regardless of the user's wishes and of system states (normal or degraded) and the latter permits the user to change his requirement to the reliability of the

required services. Between these two extremes, there exist other possible extents such as schemes supporting partial flexibility that change the reliability levels of their services only when failures appear in the system. It must be pointed out that high flexibility will certainly introduce a major degree of complexity into the control part of a scheme; this may affect the overall reliability of the system. This complexity seems a price to be paid in the design of such a flexible scheme.

Next, we will briefly discuss the problem of timing aspects. As we know, RB has a variable *response time*, in which the time necessary for generating a result is proportional to the number of failing alternates (and as will be seen below, our new scheme has similar properties). For non-real-time services (e.g., services whose specification does not include any rigourous requirement on time) this variable response time is usually acceptable. Alternatively, for real-time applications, the required response time determines the maximum number of alternates to be executed occasionally. Furthermore the usual mechanisms for dealing with time deadlines must be provided by the real-time kernel. Timers must be used for controlling the execution of variants, and can abort the execution of any alternate when deadlines expire. Work exists in the literature on applicability of software fault tolerance schemes with variable response time to real-time systems [7, 9, 12].

Rather than *independent* and *related faults* [5], here *separate* and *common mode failures* [15] of the components are considered. Related faults are either faults in the common specification, or come from dependencies in the separate designs and implementations. Two types of common mode failures may be: (i) those amongst several variants and (ii) those amongst one or several variants and the adjudicator. Independent faults usually lead to separate failures although it may happen that independent faults cause common mode failures, whereas related faults manifest under the form of common mode failures.

III. SCOP Description

The previous considerations about the main fault tolerance schemes led us to develop the Self-Configuring Optimistic Programming scheme (SCOP) for tolerating software design faults. SCOP works with any of the criteria for delivering results mentioned in Section II and its main characteristics are:

i) it uses an adjudication function generally based on comparisons of results produced by the different variants, i.e. SCOP is a syntactic scheme. This allows wide applicability: no semantic information is needed;

ii) it is an optimistic scheme that tries to execute the minimum number of variants necessary to providing a result with a sufficient assessed reliability. To do this it is organized in phases, each one involving an appropriate subset of variants. At the end of each phase, an adjudication is performed, checking if conditions for the release of a result are verified, in which case the scheme stops. This improves cost-effectiveness and avoids waste of resources;

iii) the syndrome used by the adjudication may grow at each phase containing all the relevant information collected so far. This allows SCOP, in any phase of its execution, to be as reliable as schemes that adopt the same adjudication function and use the same number of variants used by SCOP until that phase.

The SCOP scheme consists of a set **V** of N variants, designed according to the principle of design diversity, and an adjudication mechanism. Its behaviour is described by means of the following algorithm (in a Pascal-like language) which makes use of several procedures and variables whose meaning is explained below.

SCOP algorithm

```
begin i := 0; decide (f); NEF:=N;
while NEF=N do
        begin
        i:=i+1;
        select (V_i );
        execute (V_i);
        adjudication ( ∪_{K=1}^{i} O_K, NEF, res);
        end;
if NEF = E  then deliver (res) else signal (failure);
end.
```

- the procedure ***adjudication*** implements the adjudication function. It receives the syndrome (denoted by $\bigcup_{K=1}^{i} O_K$) and outputs the value of the result (if one can be selected) and one of the three possible judgments E, F and N: E is produced when the result selected is judged to be correct with the required probability; N is produced if no value can be released but it is still possible to produce the result with the required correctness probability; F is produced if it is no more possible to deliver a result with the required correctness probability and SCOP must conclude with a (detected) failure;
- the procedure ***decide*** determines how many phases can be performed;
- the procedure ***select*** selects the minimum set of variants whose execution may lead the adjudicator to a successful judgment. The variants are selected among the currently available ones, namely those variants that have not been executed in any of the previous phases. In the last executable phase all the currently available variants are selected;
- the procedure ***execute*** manages the execution of the selected variants; ***deliver*** and ***signal*** output the selected result and a failure notification respectively;
- **i** identifies the current phase, $\mathbf{V_i}$ the set of variants executed in phase i, and $\mathbf{O_i}$ the set of relevant information produced by the execution of the variants in V_i;

- **f** is the maximum number of phases that can be performed still providing timely results, computed as the lesser of $(N-|V_1|+1)$ and the number allowed by the time constraints, if any;
- **NEF** and **res** contain the outputs of the procedure adjudication.

Example

Suppose that (I) the reliability criterion to be used is that a maximum number of faulty variants are allowed in the system, let 3 be this number; (II) services are required without any timing constraint; and (III) to use an adjudication function that receives syndromes composed of the results of the executed variants and checks for at least 4 agreeing values. Accordingly, V={a,b,c,d,e,f,g} and f=4. Three examples of possible executions are given in Table II. Italic is used for agreeing results and bold for results produced in the current phase.

Phase	V_i	V	Syndrome	Judgement & result
1	{a,b,c,d}	{e,f,g}	r_a, r_b, r_c, r_d	⇒ E, r_a
1	{a,b,c,d}	{e,f,g}	$\mathbf{r_a}$, $\mathbf{r_b}$, r_c, r_d	N
2	{e,f}	{g}	r_a, r_b, r_c, r_d, r_e, r_f	⇒ E, r_c
1	{a,b,c,d}	{e,f,g}	$\mathbf{r_a}$, r_b, $\mathbf{r_c}$, r_d	N
2	{e,f}	{g}	r_a, r_b, r_c, r_d, $\mathbf{r_e}$, r_f	N
3	{g}	{}	r_a, r_b, r_c, r_d, r_e, r_f, $\mathbf{r_g}$	⇒ F

Table II. Examples of SCOP executions

In this example SCOP resembles 7VP using the majority adjudication function. SCOP algorithm terminates as soon as an agreement is found, without always requiring the execution of all the variants, as the corresponding 7VP does. SCOP is more cost-effective than NVP also in offering services inclusive of time requirements if these allow to perform more than one phase.

IV. SCOP Design Methodology

We present now a methodology for the design of a SCOP scheme. It provides the designer with a guide as to what must be done (and how to do it), once certain decisions have been taken. In designing a SCOP scheme the decisions that mostly affect the whole process are:

1) the degree of flexibility in the level of reliability of services the system is required to provide. It may range from no flexibility, with only one level, to a different level for each service. When flexibility is required the number of different reliability levels identified becomes important;
2) the estimates of the reliability of the variants to employ; these need not be the same for all the variants;
3) the criterion on which the acceptance of the results must be based. It may be some particular fault assumption or, directly, the probabilities of correctness of the results that are to be produced.

A directed **graph** containing all the information that can be derived at design time is provided. At run time the adjudication mechanism of SCOP uses this information as read only data, avoiding recomputing them at each activation, so being as simple and fast as possible. We describe, for simplicity, the design process related to the following decisions:

- a few different reliability levels R_i have been identified, the highest being R_{max};
- all the variants employed have the same figures of reliability;
- the acceptability of the results is based on assumptions on the number of faulty variants derived from the R_is.

The design begins with the computation of the minimum number N of variants, to be employed in the system, necessary to provide services with reliability R_{max}. N is derived from R_{max} and from the reliability estimates of the employable variants. The second step consists in generating all the possible syndromes for each number M ($1 \leq M \leq N$) of variants which can be executed. The syndromes are partitioned into classes, each represented by an ordered string of numbers (z_1, z_2, z_3..) with $z_1 \geq z_2 \geq z_3$...., where z_i is the number of occurrences of the i-th most frequent result in the syndrome. With M = 3, all syndromes containing just two equal results belong to the class denoted as (2,1). Then we define a relation among classes. The class S_i^M (i-th class executing M variants) is related to the class S_j^{M+1} (j-th class executing M+1 variants) if from any syndrome in S_i^M it is possible to obtain, when a further variant is executed, a syndrome belonging to S_j^{M+1}. For example, starting from all the syndromes belonging to the class (2,1), the execution of a fourth variant may lead to syndromes belonging to (3, 1). A graph can be created, in which nodes represent the classes, and oriented arcs represent this relation among classes. In Figure I we give an example of the graph that results when 5 variants are designed.

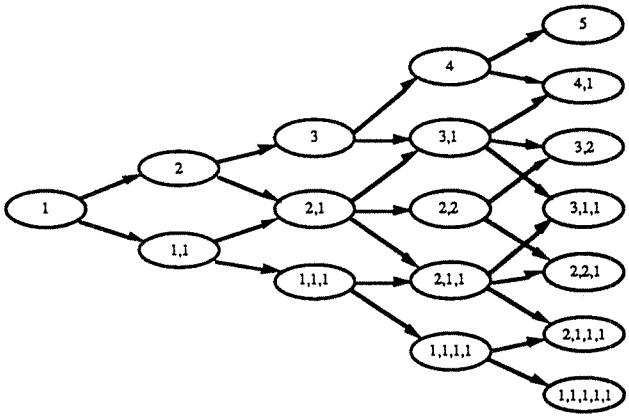

Figure I. Syndromes graph for N=5.

The subsequent steps provide information to be associated to each node in the graph. The first item of information concerns the reliability assessment of the most frequent result for each class. In our case this is the range in the number of faulty variants, if any, that may be present still allowing to select this result as the correct one. Each node in the graph will have a label carrying this information. For example node (3) in the graph representing three agreeing results

out of the execution of three variants, contains the information that the most frequent result is acceptable if [0, 2] variants may have failed. There are nodes for which no result can be guaranteed as correct under any circumstances, for example (1,1). The general rule for providing this information is as follows. For each node in the graph obtained from the execution of M variants, labelled with a string of j numbers (z_1, z_2, .., z_j), let $F_i = M - z_i$, i=1..j, the number of failures in the system for the i-th result to be provably correct, and let $F_{j+1} = M$ represent the case in which all the variants have failed. $F_1 \leq F_2 \leq ..\leq F_j < F_{j+1}$. Let k be the maximum number of failures admitted in the system. The result occurring z_1 times may be correct if $k \geq F_1$ since if $k < F_1$ the number of failures experienced is too high and the result not acceptable. Moreover, this result is the only correct one if $k < F_2$ since if $k \geq F_2$ the result occurring z_2 times may also be correct. Therefore, if $F_1 < F_2$ the information recorded is the interval $[F_1, F_2 - 1]$; if $F_1 = F_2$, the information recorded is simply F_2 and no result can be guaranteed as correct for any number of faults.

Information related to the set of different reliability levels $\{R_i\}$ identified can be derived now. For each of these levels R_i, each node in the graph contains information about (I) the acceptability of the most probably correct result for the syndromes in the class and (II) if this is not acceptable, the indication of further actions to perform at run time. Now we show how this information is derived. Let $[0, K_i]$ be the range of failures admitted in the fault assumptions representing reliability level R_i. With respect to R_i, the classes can be classified as either *end-classes*, *non end-classes* or *failure classes*. The classes for which the interval $[F_1, F_2 - 1]$ has been defined are:

- end-classes if $F_1 \leq K_i \leq F_2 - 1$
- non end-classes if $F_2 - 1 < K_i$
- failure classes if $F_1 > K_i$.

Those labelled with only F_2 are:

- non end-classes if $F_2 - 1 < K_i$
- failure classes if $F_2 > K_i$.

End-classes correspond to the success of the scheme, i.e. one result can be selected as being provably correct. Non end-classes are those for which the scheme has not yet reached success but where success is still possible: further variants must be executed. Failure classes are those representing the (detected) failures of the scheme: too many variants have failed. The nodes in the graph will contain the corresponding mark in the set {E, N, F}. Nodes representing end-classes and failure classes contain sufficient information, while for non end-class nodes the minimum number X of further variants to execute in a next phase for possibly reaching an end-class must be provided. Given a syndrome belonging to a non end-class ($F_2 - 1 < K_i$), let X further variants be executed and (optimistically) suppose that all agree with the already most frequent result, obtaining a syndrome belonging to (z_1+X, z_2,....). If we designate F_i' the value of F_i for this class, $F_1' = M + X - (z_1 + X) = M - z_1 = F_1 \leq K_i$ (since this was a non-end class) and $F_2' - 1 = M + X - (z_2) - 1 = F_2 - 1 + X$.
The minimum X to satisfy the condition for end-classes ($F_1' \leq K_i \leq F_2' - 1$) is:
$K_i = F_2 - 1 + X$ and then $X = K_i - (F_2 - 1)$.

The last step consists in determining, for each reliability level R_i, the number of variants for the first phase. The dummy class obtained executing 0 variants, (0) can be added to the graph. It is a non-end class since F_1=0 and F_2=0, and the rule just described can be applied: $X = K_i - (F_2 - 1) = K_i + 1$.

Following such an analysis, SCOP can be put now into operation. In the interesting case we have considered, the adjudication mechanism of the scheme needs just to read in the graph the information (N, E or F) related to the required reliability level for a service. The work to be performed at run time is limited to following the development of the execution on the graph, i.e. to move from one node to another according to the observed syndrome.

An example is now given of all the information which can be provided. The number of variants available is 5 and two different reliability levels have been identified, $R_{max} = R_1$, in which up to two faulty variants are permitted in the system, and R_2, in which only one faulty variant is permitted. The information are presented here in a table, but in general the scheme will maintain them in a graph. The row of the (0) group contains the number of variants to start with. E, N, F represent end-class, non end-class and failure class marks respectively and •• is used for cells without any information.

			R_1 : [0, 2]		R_2 : [0, 1]					R_1 : [0, 2]		R_2 : [0, 1]	
Groups	$[F_1, F_2-1]$	F_2	Mark	X	Mark	X	Groups	$[F_1, F_2-1]$	F_2	Mark	X	Mark	X
5	[0,4]	••	E	••	E	••	2,1,1	[2,2]	••	E	••	F	••
4,1	[1,3]	••	E	••	E	••	1,1,1,1	••	3	F	••	F	••
3,2	[2,2]	••	E	••	F	••	3	[0,2]	••	E	••	E	••
3,1,1	[2,3]	••	E	••	F	••	2,1	[1,1]	••	N	1	E	••
2,2,1	••	3	F	••	F	••	1,1,1	••	2	N	1	F	••
2,1,1,1	[3,3]	••	F	••	F	••	2	[0,1]	••	N	1	E	••
1,1,1,1,1	••	4	F	••	F	••	1,1	••	1	N	2	N	1
4	[0,3]	••	E	••	E	••	1	[0,0]	••	N	2	N	1
3,1	[1,2]	••	E	••	E	••	0	••	••	N	3	N	2
2,2	••	2	N	1	F	••							

Table III. Information for SCOP with 5 variants.

The methodology also offers to the designer an indication as to what has to be done about the reconfiguration of SCOP. A variant affected by an hard fault [14] must be isolated from the scheme and the scheme itself reconfigured. The necessary steps are:

1) to extract the faulty variant from the set of available ones;
2) to erase all the leaf nodes (representing syndromes of N results) from the graph;
3) for each level R_i, all non end-class nodes must be checked. If the reduced number of available variants does not allow any more for reaching end-classes, the node must be classified as a failure class.

The methodology consists essentially of the same steps whatever choices are made. The algorithms presented depend on the decisions taken, and, if different choices are made, the algorithms to be performed at each step may be much different. In the following some examples are given, not representing all the necessary changes to be made coping with different choices, but just to suggest how particular steps have to be reconsidered. If the criterion for delivering results is based on its probability of being correct, the information that we represented in the graph with $[F_1, F_2 - 1]$ must be substituted with the probability of being correct $Pmax(S_i^M)$ of the value with the highest probability of correctness in S_i^M that can be determined from the figures of the variants [8]. If variants with different reliability figures are chosen, the most significant change involves the construction of the graph. A fixed ordering among variants may be defined, meaning that variants will always be executed in the same order (variant i will be executed before or together with variant i+1, but never in a subsequent phase). Then the graph must be constructed according to a different partition, containing an increased number of classes, that considers the ordering of variants. For example, the syndromes belonging to class (2, 1) of the partition in our example, must be split in three classes distinguishing which variant gave the result in disagreement. If the required flexibility implies an impracticable number of identified reliability levels, all the information relative to these levels must be computed at run time for each service and the graph will contain only the information on the reliability assessment of the 'best' result for each class. If no flexibility is required, just one reliability level is considered.

For each phase of the design, the set of algorithms to apply, each one related to a different set of decisions, can be defined. It is, therefore, a clerical work to build an automatic tool for designing SCOP and integrate it in a programming environment for software development. Software designers may then use it and automatically derive instances of SCOP. They just have to take the proper decisions and are not required to perform the necessary analysis for any instance of SCOP they want to design.

V. Evaluation

In this section, we analyse the SCOP scheme in a configuration that makes it similar to other approaches and evaluate its reliability and cost-effectiveness compared with the other main schemes. In [3] the architectures of RB, NVP and NSCP, as used in order to tolerate a single software failure, have been analysed. We shall use some of these results, and will exploit that framework for considering the software redundancy needed to tolerate two failures. Four schemes are considered: SCOP and NVP using 5 variants, the former adopting a threshold adjudication requiring 3 agreeing results and the latter the usual majority adjudication, RB using 1 primary and 2 alternates and NSCP using 6 variants organized as 3 self-checking components. The following analysis considers only value correctness, and assumes no timing constraints (the usual way in which reliability analyses have been made in the literature [3, 15]).

Reliability evaluation

Basic Assumptions and Notation

Let X indicate one of the four schemes considered: SCOP, NVP, RB, NSCP.

(i) during the execution of scheme X four types of component failures can be observed:
   1) a separate failure of the adjudicator, with probability $q_{A,X}$;
   2) a common mode failure between the variants and the adjudicator, with probability $q_{VA,X}$;
   3) a separate failures of one or more variants, each one with probability $q_{I,X}$;
   4) a common mode failure involving S variants, with probability $q_{SV,X}$;

(ii) the probability of separate failure is the same for all the variants;

(iii) only a single failure type may appear during the execution of the scheme and no compensation may occur between the errors of the variants and of the adjudicator;

(iv) the failure of the adjudicator, alone or together with failures of the variants, is never detected; the separate failures of variants are always detected and common mode failures of variants are undetected only if **S** is such that the adjudicator, although correct, will chose the incorrect result of these variants.

We denote with $q_{UV,X}$ the sum of the probabilities of the undetected common mode failures among variants for scheme X; e.g. when N=5, $q_{UV,NVP}$ is the sum of the probabilities of the common mode failures among 3, 4 and 5 variants. Let $q_{UF,X}=q_{A,X}+q_{VA,X}+q_{UV,X}$ denote the probability of an undetected failure, $q_{DF,X}$ the probability of a detected one and $Q_{R,X} = q_{UF,X} + q_{DF,X}$ the probability of failure of the X scheme. The expressions for each $Q_{R,X}$ can be obtained using a Markov approach. In the following the indication of the scheme will be omitted when clear from the context.

Detailed Reliability Model

Software failures can manifest themselves only when software is executed. Thus, a simple behaviour model can be described as in Figure II. If the departure rate from state I is λ, then the reliability of the X approach can be evaluated by: $R_X(t) = e^{-\lambda Q_{R,X} t}$.

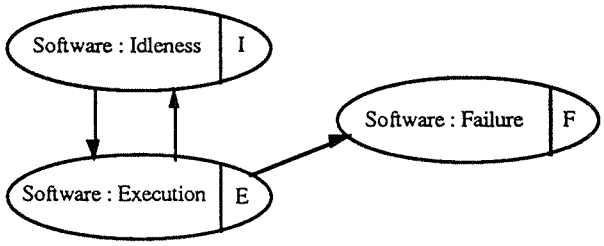

Figure II. A brief behaviour model.

In the following, we will only show the Markov chain for SCOP. For the other schemes, similar models can be built. In Figure III, state E is the execution state of software. States from A1 to A11 correspond to the execution of the adjudicator. Respectively,

(1) state A1 indicates that the three acting variants present three agreeing correct results;
   $p = 1 - 3q_I - 9(q_I)^2 - 10(q_I)^3 - 5(q_I)^4 - (q_I)^5 - 9q_{2V} - 10q_{3V} - 5q_{4V} - q_{5V} - q_{AV}$;

(2) state A2 indicates a separate failure of one of the three acting variants;

(3) state A3 shows two separate failures of two out of five acting variants;

(4) states A4, A5 and A6 indicate multiple separate failures;

(5) state A7 represents a common mode failure of two among five variants;
(6) states A8, A9 and A10 correspond to a common mode failure of three or more variants;
(7) state A11 implies a related failure in the adjudicator and the variants.

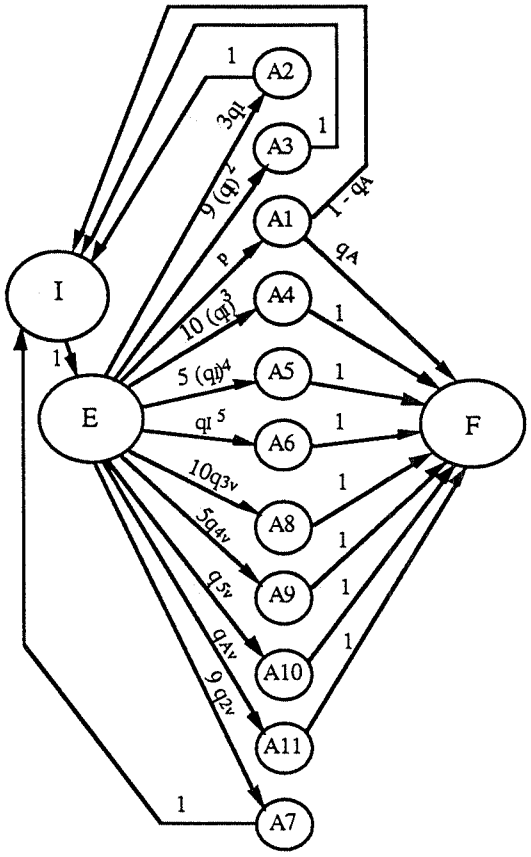

Figure III. The SCOP Model.

From the above model, we obtain:
$Q_{R,SCOP}=pq_A+10(q_I)^3+5(q_I)^4+(q_I)^5+10q_{3V}+5q_{4V}+q_{5V}+q_{AV}$;
A close but pessimistic approximation of this is:
$Q_{R,SCOP} = q_A+10(q_I)^3+5(q_I)^4+(q_I)^5+10q_{3V}+5q_{4V}+q_{5V}+q_{AV} = q_{UF1}+10(q_I)^3+5(q_I)^4+(q_I)^5$;

We conclude, using a similar method, that:
$Q_{R,NVP} = q_{UF2}+10(q_I)^3+5(q_I)^4+(q_I)^5$;
$Q_{R,RB} = q_{UF3}+(q_I)^3+ q_{3V}$;
$Q_{R,NSCP} = q_{UF4}+8(q_I)^3+12(q_I)^4+6(q_I)^5+(q_I)^6$;

Note that the probabilities associated with the adjudicators may be significantly different due to the availability of various complicated adjudicators [8]. The SCOP adjudication mechanism is in charge of deciding also the number of variants to execute in each phase, activity that increases its complexity. As shown in Section IV, these decisions may be taken both at design time and at run time. So, we distinguish different levels of complexity for the SCOP adjudication, depending on the degree of dynamism allowed for it. One extreme is the *static adjudication*

obtained taking all the decisions at design time, as in the design developed in the previous section. The other extreme, the *dynamic adjudication*, is obtained if all the decisions are taken at run time in accordance with different design choices. Obviously, the complexity of the adjudication increases from the static to the dynamic one. The following table summarizes the specific expressions for $q_{UF,X}$.

$q_{UF,X}$	SCOP	NVP	RB	NSCP
$q_{A,X}$	q_A(stat./dynam.)	q_A(voter(N))	q_A(AT)	q_A(comparator)
$q_{VA,X}$	$q_{VA,SCOP}$	$q_{VA,NVP}$	$q_{VA,RB}$	$q_{VA,NSCP}$
$q_{UV,X}$	$10q_{3V}+5q_{4V}+q_{5V}$	$10q_{3V}+5q_{4V}+q_{5V}$	0	$q_{2V}+18q_{3V}+14q_{4V}$ $+6q_{5V}+q_{6V}$

TABLE IV. Probability of Undetected Failures.

Some conclusions can be derived:

1) RB seems to be the best, but the AT is semantics or application-dependent reducing the possibility of design diversity between variants and AT, such that $q_{VA,RB}$ (i.e., the probability of related failures between AT and alternates) may vary dramatically.
2) When SCOP employs static adjudication, as for most of the time it needs to execute only the first phase (with M<N variants), $q_{VA,SCOP} < q_{VA,NVP}$ and $q_{A,SCOP} < q_{A,NVP}$, since voter(M) used by SCOP is simpler than voter(N) used by NVP. So, in this case, $Q_{R,SCOP} < Q_{R,NVP}$.
   However, when SCOP uses a dynamic adjudication, $q_{A,SCOP} > q_{A,NVP}$, but still, as for most of the time only the first phase involving M<N variants is needed, $q_{VA,SCOP} < q_{VA,NVP}$. Then, it may be reasonable to conclude that $Q_{R,SCOP} \cong Q_{R,NVP}$.
3) NSCP suffers from related failures among variants in spite of its low $q_{A,NSCP}$ and $q_{VA,NSCP}$.

Consumption of Resources.

Let k be the maximum number of faults to be tolerated, d the number of detected faults ($d \leq k$) and $T = T_V + T_A$ be the time necessary for the execution of a complete phase (T_V for the versions and T_A for the adjudication function). The following table reports some results about resource consumption, again referring to the architectures used in the reliability analysis, without taking into account specifications with timing constraints. Moreover, we assume perfect adjudicators. NVariants indicates the total number of variants execution necessary to the scheme to complete its execution.

Schemes	NVariants worst	NVariants average	TIME worst	TIME average
SCOP	k+1+d	$\cong$(k+1)	T+dT	$\cong$T
NVP	2k+1	2k+1	T	T
RB	1+d	$\cong$1	T+dT	$\cong$T
NSCP	2(k+1)	2(k+1)	$T+dT_{switch}$	$\cong$T

TABLE V. Comparison of resource consumption.

1) SCOP will perform only the first phase if (k+1) agreeing results are produced. This happens if (I) a common mode failure involves all the (k+1) variants; or (II) all the variants produce the same correct result. Events like the successes or failures of individual variants, when executed together on the same input, are usually not independent but positively correlated [13]. This factor determines a probability of observing event (II) higher than might be expected assuming independence. Let p_V be the probability that a single variant gives a correct result, then P(SCOP stops at the end of the first phase) > P(k+1 correct values) > p_V^{k+1} (probability computed assuming independence). Experimental values of p_V ([13]) are sufficiently high to state that SCOP almost always gives the same fast responses as NVP. RB has a higher probability of stopping at the first phase than SCOP (SCOP requires k+1 agreeing results, RB only that the primary be correct).
2) The worst case when (T+dT) is the time necessary to SCOP to conclude has a very rare probability of occurrence. In fact, it occurs only when the first phase ends with k agreeing results and one different, the (k+1)-th result necessary for success is produced during the (d+1)-th phase, and all the variants run from the second to the d-th phase (one for each phase), produce a different result.

In delivering services with time constraints for which the maximum number of allowable phases is f ($f \leq k+1$), SCOP's worst case in time derivation is T*f. Note, however, that this limit on the number of phases impacts the average usage of variants only if f=1, in which case the execution of all the variants is required, otherwise the first phase always involves only (k+1) variants. The basic RB cannot be applied when $f<k+1$, as k+1 phases is the time duration RB requires to assure a correct result under the assumption of k faults. Parallel implementations of RB exist and they allow to cope with time limits, but at the cost of a higher number of variants executed.

An example: SCOP compared with NVP

Table IV shows that NVP and SCOP, both with 5 variants, have approximately the same reliability. They have exactly the same reliability under the assumption of perfect adjudicators (refer to the row relating to $q_{UV,X}$ in Table IV). Now, adding this assumption to those previously made, we generalize this result, in an informal but intuitive way, for the same organization of NVP and SCOP considering N=2k+1 (k faults to be tolerated). Then we show that SCOP improves cost-effectiveness and give a numerical example.

Reliability

SCOP and NVP have the same reliability because they always make the same choice in delivering a result. This result may be the result selected in cases of success and undetected failure, or an exception when a failure is detected. Two cases must be distinguished. If SCOP stops at the end of the first phase, it means that the k+1 variants executed have produced the same result, and so this has been judged to be the correct one. As k+1 constitutes a majority among 2k+1 variants, this result is also the majority result for NVP, regardless of the values of the remaining k variants. If SCOP does not stop at the end of the first phase, further phases are

performed according to the policy described. At the end of each phase the adjudicator of SCOP checks whether there are at least k+1 equal results among all those collected until the current phase. If this agreement exists, SCOP and NVP will still make the same choice, whatever the phase in which the SCOP scheme terminates. The above conclusions consider that a parallel or sequential execution of a set of variants always return the same syndrome. In reality, the observed syndromes may be different in the two cases, but an evaluation of such a difference is very difficult to perform.

Cost-effectiveness

We now focus our attention on resource consumption and response time. Let us organize the execution of SCOP in two phases, with (k+1) variants running during the first phase and the remaining k during the second phase. Let T be the time necessary for the execution of a complete phase (as defined before). We make an explicit example assuming k=2 and $p_V = (1-10^{-4})$ which is the average reliability of the versions resulting from the experiment in [13]. The probability that SCOP executes only the first phase can be bounded with p_V^{k+1} as discussed before. The table shows the related figures of SCOP and NVP.

	General case		Example	
Average	SCOP	NVP	SCOP	NVP
No. of variants executed	$(k+1)+(1-p_V^{k+1})*k$	2k+1	3.0006	5
No. of adjudications	$1 + (1 - p_V^{k+1})$	1	1.0003	1
Time consumption	$T+ (1 - p_V^{k+1}) * T$	T	1.0003 T	T

Table VI. Comparison of resources used by SCOP and NVP.

The maximum response time is 2T. The run time overhead required by SCOP with N=2k+1 variants is much closer to that required by (k+1)VP (which resists k/2 component failures) rather than the overhead required by (2k+1)VP. Therefore we conclude that SCOP is more cost-effective than NVP.

VI. Conclusions

In this paper we have introduced the Self-Configuring Optimistic Programming scheme for software fault tolerance. SCOP has been defined with the aim of achieving a good trade-off between the different characteristics that matter in software fault tolerance. It is (or can be) parametric to the reliability of the services which it must provide. While providing services with the requested 'quality', the scheme tries to minimize the amount of redundancy actually used, being in most cases very successful and resulting in very good cost-effectiveness.

We have shown that SCOP has reliability figures of the same oreder to those of the other schemes and, if able to deliver a result at the end of the first phase, meets the optimality in resources consumption and time necessary for delivering results with the required reliability. The actual effort, in terms of both the number of variants executed and time, depends on (and is justified by) the number of faults experienced. Together with its cost-effectiveness, the main

advantage of SCOP is that it is parametric to the degree of reliability required for the services. This represents a major novelty in software fault tolerance and results in a very high degree of flexibility thus making SCOP a proper tool for gracefully degradable systems. Adopting SCOP, users are able to decide at run time, according to the available resources, what is to be sacrificed: a) the throughput of the system as number of services delivered per time unit; or b) the time necessary for service delivery; or c) the reliability with which some selected services are provided.

Beside the basic work presented in this paper much remains to be done. The reliability of the result to be provided as the criterion for delivering services seems to lead to a better approach to software fault tolerance. To develop this approach a better understanding is necessary of the relations between the fault tolerant systems reliability and the reliability associated to the individual services they provide. In this paper SCOP has been evaluated according to software aspects solely. Its behaviour and performance characteristics, when used for tolerating also hardware faults, thus considering hardware architectures and components, must be still evaluated. A third line of work is directed towards extensions of the SCOP scheme for example, (I) generality, i.e. to evaluate if some of its possible configurations may make SCOP equivalent to other schemes used in software fault tolerance, and (II) nesting, i.e. usage of instances of fault tolerance schemes as variants.

References

[1] T. Anderson, "Resilient Computing Systems," Collins Professional and Technical Books, 1985.

[2] T. Anderson, "A Structured Decision Mechanism for Diverse Software," in Proc. 5-th Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, California, 1986, pp. 125-129.

[3] J. Arlat, K. Kanoun and J.C. Laprie, "Dependability Modelling and Evaluation of Sofware Fault-Tolerant Systems," IEEE TC, Vol. C-39, pp. 504-513, Apr. 1990.

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

[5] A. Avizienis and J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," IEEE Computer, Vol. 17, pp. 67-80, Aug. 1984.

[6] A. Avizienis and J.C. Laprie, "Dependable Computing: from Concepts to Design Diversity," Proc. of the IEEE, Vol. 74, pp. 629-638, May 1986.

[7] R.H. Campbell, K.H. Horton and G.G. Belford, "Simulation of a Fault Tolerant Deadline Mechanism," in Proc. FTCS-9, Madison, Wisconsin, 1979, pp. 95-101.

[8] F. Di Giandomenico and L. Strigini, "Adjudicators for Diverse Redundant Components," in Proc. SRDS-9, Huntsville, Alabama, 1990, pp. 114-123.

[9] H. Hecht, "Fault Tolerant Software for Real Time Applications," ACM Comp. Surveys, Vol. 8, pp. 391-407, Dec. 1976.

[10] B.W. Johnson, "Design and Analysis of Fault Tolerant Digital Systems," Addison-Wesley Pub. Co., 1989.

[11] H.K. Kim, "Distributed Execution of Recovery Blocks: an Approach to Uniform Treatment of Hardware and Software Faults," in Proc. 4-th Conf. on Distributed Computing Systems, 1984, pp. 526-532.

[12] K.H. Kim and H.O. Welch, "Distributed Execution of Recovery Blocks: an Approach for Uniform Treatement of Hardware and Software Faults in Real-Time Applications," IEEE TC, Vol. C-38, pp. 626-636, May 1989.

[13] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," IEEE TSE, Vol. SE-12, pp. 96-109, Jan. 1986.

[14] J.C. Laprie, "Dependability: a unifying concept for reliable computing and fault tolerance," in "Dependability of Resilient Computers", T. Anderson Ed., BSP Professional Books, 1989, pp. 1-28.

[15] J.C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Definition and Analysis of Hardware and Software Fault-Tolerant Architectures," IEEE Computer, Vol. 23, pp. 39-51, Jul. 1990.

[16] 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. FTCS-17, Pittsburgh, 1987, pp. 116-121.

[17] P.A. Lee and T. Anderson, "Fault Tolerance: Principles and Practice," Prentice-Hall, 2nd Ed. 1990.

[18] F. Lombardi, "Optimal Redundancy Management of Multiprocessor Systems for Supercomputing Applications," in Proc. 1st. Int. Conf. on Supercomputing Systems SCS-85, St. Petersbourgh, Fl., 1985, pp. 414-422.

[19] P.R. Lorczak, A.K. Caglayan and D.E. Eckhardt, "A Theoretical Investigation of Generalized Voters for Redundant Systems," in Proc. FTCS-19, Chicago, 1989, pp. 444-451.

[20] B. Randell, "System Structure for Software Fault Tolerance," IEEE TSE, Vol. SE-1, pp. 220-232, 1975.

[21] R.K. Scott, J.W. Gault and D.F. McAllister, "Fault-Tolerant Software Reliability Modeling," IEEE TSE, Vol. SE-13, pp. 582-592, 1987.

[22] G.F. Sullivan and G.M. Masson, "Using Certification Trails to Achieve Software Fault Tolerance," in Proc. FTCS-20, Newcastle, 1990, pp. 423-431.

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