Dynamic Adjustment of Dependability and Efficiency in Fault-Tolerant Software

Jie Xu[1] Andrea Bondavalli[2] Felicita Di Giandomenico[3]

[1]University of Newcastle-upon-Tyne [2]CNUCE-CNR [3]IEI-CNR

Abstract. In this paper we discuss the problem of attaining a dynamic compromise between using redundancy to improve software dependability and limiting the amount of redundancy so as to avoid unnecessary inefficiencies. A scheme, called *self-configuring optimal programming* (SCOP), is developed. SCOP attempts to reduce the resource cost of fault-tolerant software, both in space and time, by providing designers with a flexible redundancy architecture in which dependability and efficiency can be adjusted dynamically at run time. A design methodology is proposed to introduce support techniques for such dynamic adjustment. Our scheme also suggests a general control framework into which design diversity, data diversity and multiple copies can be incorporated selectively. A detailed dependability and efficiency evaluation shows that SCOP can achieve the same dependability level as those of other existing schemes, while making more efficient use of available resources.

1 Introduction

Dependability of a computing system refers to the quality of the service delivered by the system such that reliance can be justifiably placed on this service, and serves as a generic concept encompassing notions of reliability, availability, safety, security, performance, etc. (see [Laprie, 1995, chapter1, internal reference] and [Anderson, 1985 #11]). A dependable 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 large number of techniques, including *fault tolerance* techniques intended to cope with the effects of *faults* and avert the occurrence of *failures* or at least to warn a user that *errors* have been introduced into the state of the system. The provision of tolerance to anticipated hardware faults has been a common practice for many years, whereas a relatively new development is the techniques for tolerating unanticipated faults such as design (typically software) faults. (To avoid confusion in terminology, we will use in the following the term “fault-tolerant software” to characterize the software system that primarily treats software faults but may mask some hardware-related faults at the software level.)

Because software faults are permanent in nature, software fault tolerance in principle requires redundancy of design and/or data [Ammann, 1988 #2] although some software faults may be masked just by retry and message reordering [Huang, 1993 #12]. Design redundancy, or design diversity, is the approach in which the production of two or more components of a system is aimed at delivering the same service through independent designs [Avizienis, 1986 #13]. The components, produced through the design diversity approach from a common service specification, are called variants. Tolerance to design faults necessitates an adjudicator [Anderson, 1985 #11] or a

decision algorithm that provides an (assumed to be) error-free result from the execution of variants. The major advantage of design diversity is that dependable computing does not require the complete absence of design faults, but only that they should not produce similar errors in variants. Classical techniques for tolerating software faults are mostly based on some form of design diversity, including recovery blocks [Randell, 1975 #14], *N*-version programming [Avizienis, 1977 #4], *N* self-checking programming [Laprie, 1987 #15], *t*/(*n*–1)-variant programming [Xu, 1991 #17], and some intermediate or combined techniques [Kim, 1984 #7][Scott, 1987 #9].

There are two fundamental problems that need attacking. First, traditional approaches to software fault tolerance can be very costly. The cost of developing the variants and adjudicator may be many times more than that of programming a single version [Laprie, 1990 #16]. Despite of high cost, design diversity still has some difficulties in ensuring a routine-based improvement in software dependability. The interested reader is referred to [Eckhardt, 1991 #19] for details of such discussion. Experience has shown that some techniques such as rollback-and-retry may mask a range of software faults that produce transient effects [Huang, 1993 #12]. The object-oriented programming paradigm has also shown much promise in reducing the development cost of fault-tolerant software through inheritance and polymorphism mechanisms [Xu, 1994 #18].

Secondly, most of the methods for software fault tolerance are not particularly efficient, as will be discussed in Section 2, though *efficiency* remains an important aspect of software quality. Possible resolutions of the efficiency problem are the subject of this paper. Efficiency is defined in this paper as the good use of system and hardware resources, such as processors, internal and external memories, and communication devices [Meyer, 1988 #20]. Since the kind of applications which require software fault tolerance are often also likely to have stringent efficiency requirements, a good use of the available resources, both in space (hardware) and time (repetition), is highly desirable. In fact, the pursuit of increased efficiency in military computer-based systems, particularly in the economic and timely use of resources, is more apparent today than at any time in the past [Mills, 1993 #21].

2 Tradeoff between Software Dependability and Efficiency

Recovery blocks (RB) are the first scheme designed to provide software fault tolerance. 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. The 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. The execution time of a recovery block is normally that of the first variant, acceptance test, and the operations required to establish and discard a checkpoint. This will not impose a high run-time overhead unless an error is detected and backward recovery required. In this regard, RB is highly efficient. Limitations of the RB method are mainly related to its acceptance test. This test is usually derived from the semantics of a given application, and close dependency between the test and variants may impact dependability of the whole system. Such a test will also introduce a run-time overhead which could be unacceptable if the test is complex. However, the development of simple, effective acceptance tests is a difficult task.

The next three approaches avoid use of an acceptance test by taking advantage of parallel execution of multiple variants and result's comparison (although sequential execution is conceptually possible just as parallel execution of RB alternates is possible). *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. *N*-self-checking programming (NSCP) provides fault tolerance through parallel execution of *N* self-checking components. Each self-checking component is constructed from a pair of variants plus a result comparator (or from a variant associated with an acceptance test). One of them is regarded as the active component, and the others as "hot" standby 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 based on the system diagnosis technique developed for hardware. This approach uses a particular diagnosability measure, *t*/(*n*–1)-diagnosability, and it can isolate the faulty variants within a set of at most (*n*–1) variants. By applying a diagnosis algorithm to results produced by variants, a result is selected (the one that, among those produced, has the highest probability of being correct) as the system output.

The adjudication mechanisms used in these three schemes are usually based on result comparison and independent of semantics of the applications, thus the probability of common mode failure between the adjudicator and the variants is relatively low in these schemes. When variants are executed in parallel, NVP, NSCP and *t*/(*n*–1)-VP may have a fixed response time (without repetition), thereby guaranteeing timely responses in the presence of faults. However, these architectures utilize redundancy in a static manner and always execute all of their variants regardless of the normal or abnormal state of the system. They are intended to tolerate the maximum number of faults that may be present in the system; but, since such a "worst case" rarely happens, the amount of resources consumed is often higher than necessary. In this sense, they are not efficient.

All the fault tolerance approaches require some extra space or extra time, or both. Figure 1 summarises space-time overheads in software fault tolerance schemes. The space is defined as the amount of hardware (e.g. the number of processors) needed to support parallel execution of multiple variants. The time is viewed here as the physical time needed to execute one or more variants sequentially. Note that efficient use of the available resources generally requires dynamic management and conditional execution of the software variants. This should come with a dynamic tradeoff between full parallel execution and totally sequential execution of the variants, as shown in the diagram. However, the majority of software fault tolerance schemes do not attempt to provide such a dynamic space-time tradeoff though it can in fact be achieved and so we would argue it should be provided. As a possible solution, we propose in the next section a new scheme, called Self-Configuring Optimal Programming (SCOP), which improves the efficiency of fault-tolerant software by diminishing the waste of resources without compromising software dependability.

Although the techniques discussed above generally require the use of diverse designs, multi-variant programming cannot always guarantee a signficant dependability improvement in a cost effective manner [Eckhardt, 1991 #19]. The work on using simple retry of programs to mask the effects of the faults that cause transient errors [Gray, 1993 #22] seems to fit practical experience, but it is less complete and its effectiveness may be a matter of luck. Using data diversity to tolerate design faults in

software systems [Ammann, 1988 #2] might provide a more cost effective alternative though such a technique is not generally applicable either. Given the existing problems in the area of software fault tolerance, SCOP is to be a general and conceptual scheme for coping with both dependability and efficiency. In order to tolerate software faults (and some hardware-related faults), an instance of our scheme may employ an application-specific strategy for masking the effect of faults, such as multiple versions of software, diversity in data space, or simple retry of programs, much depending upon special application requirements and considerations of cost effectiveness. Its mechanisms for combining dependability with efficiency rely on highly dynamic control and adaptive redundancy management, but basically independent of ways of redundancy for masking software faults.

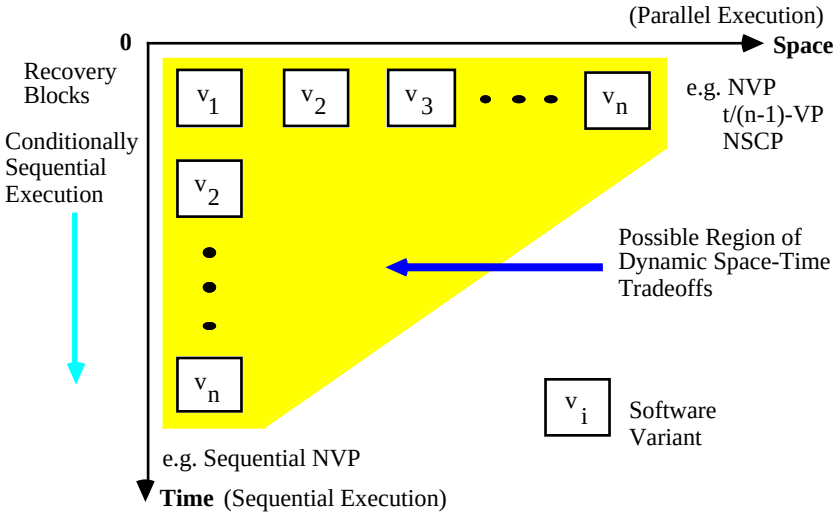

Fig. 1. Space and time redundancy in software fault tolerance.

In order to use redundancy in a dynamic or conditional manner, a scheme has to decide, at appropriate intermediate points of its execution, which of the following three execution states has been reached: i) *End-state E* — a result exists that meets the stated *delivery condition* and can thus be delivered; ii) *Non end-state N* — there is no result that meets the condition, but it is still possible to obtain such a result if further redundancy is employed; or iii) *Failure state F* — there is no further possibility of producing a result that meets the condition. In reality, conditions for delivering a result are usually embedded in the adjudicator explicitly or implicitly. Note that different conditions in principle have different fault coverages though some conditions seem to be very similar. This influences both dependability and resource consumption. In further consideration of efficient use of the available resources, our proposed scheme is devised to admit different delivery conditions, thus becoming parametric to the level of fault tolerance.

Dynamic redundancy for the purpose of space-time tradeoff is a classical idea, e.g. Duplicated Configuration with a Spare and NMR with Spares used in hardware [Johnson, 1989 #23]. Similar schemes have been applied to redundancy management in multiprocessors [Lombardi, 1985 #25] or in distributed computing [Babaoglu,

1987 #24]. However, the existing proposals take only hardware faults into account, such as processor and communication faults. Our major concern in this paper is software faults and fault tolerance at software levels. We focus our attention on both dependability and efficiency, searching for a systematic way of using the minimum amount of hardware and time resources to attain the *required* software dependability. It is particularly emphasized here that a quality fault-tolerant software should be obligated to behave as precisely required (e.g. by different delivery conditions), but it should not be liable for anything outside of the specified requirement.

For non-real-time services whose specifications exclude any rigorous requirement on time, schemes based on dynamic redundancy would be the best technical and economical choice. However, in a real-time environment the uncontrolled application of time redundancy can lead to a delay in the production of output information and will result in such information being classed as invalid if deadlines are missed. The maximum possible delay in our approach must be determined carefully according to the response time required. Related work exists in the literature on the applicability of the schemes associated with time redundancy to real-time systems [Campbell, 1979 #5][Hecht, 1976 #6][Kim, 1989 #8]. Integration of fault tolerance and real-time issues is addressed thoroughly in [Bondavalli, 1993 #27].

3 Self-Configuring Optimal Programming

3.1 Basic Architecture

The SCOP scheme consists of a set of software variants, $V=\{v_1, v_2, ..., v_n\}$, designed according to the principle of design diversity, an adjudication mechanism, and a controller that coordinates dynamic actions of the architecture. The main characteristics of SCOP include:

i) *Dynamic Use of Redundancy*: SCOP always tries to execute the least number of variants strictly necessary for providing a result that meets the stated delivery conditions. To do this it organizes the execution of variants in phases, dynamically configuring a currently active set (CAS) V_i, a subset of V, at the beginning of the ith phase. An adjudication is made after the execution of V_i in order to check if conditions for the release of a result are satisfied. The result will be output immediately and any further phases and actions will be ended once these conditions are met.

ii) *Growing Syndrome Space*: A syndrome is defined here as a set of information used by an adjudicator to perform its judgement as to the correctness of a result, in general involving those results produced by variants. The syndrome information in SCOP is accumulated with the increase of phases. All the results produced and the additional information collected so far are employed to support the selection of a correct result. This benefits dependability.

iii) *Flexibility and Efficiency*: SCOP can be designed to obey different delivery conditions. Since the different conditions will usually have different fault coverages, SCOP is therefore able to provide different levels of dependability. The different conditions may be dynamically chosen by different applications, according to their degrees of criticality, or by the same application at different times, according to the degradation of the system. The initial CAS V_1 in phase one can be determined and flexibly changed with respect to different delivery conditions. Furthermore, the set V_i in the ith phase (i>1) can be constructed at

run time based on information about the state of the system, so making efficient utilization of the available resources.

iv) *Generality*: Software variants used by SCOP can be simple copies of a program without using design diversity to cope with faults that lead to transient errors. These copies may be further associated with data diversity to mask the effects of another class of software faults.

```
begin
i:= 0;                                      {index of the current phase, set to 0}
State_mark := N;                            {set current state as non end-state}
Si = {};                                    {set syndrome as empty}
C := one of { delivery conditions };        {set required delivery condition}
decide(max_phase);                          {based on time constraints}
while State_mark = N and                    {while current state is non end-state and
   i < max_phase do                         current phase < maximum allowed}
   begin
   i := i+1;                                {start new phase}
   configure(C, Si-1, i, Vi);               {set new Currently Active Set}
   execute(Vi, Si);                         {execute and obtain new syndrome}
   adjudicate(C, Si, State_mark, res);      {set new state mark and select result}
   end;
if State = E                                {current state is end-state or failure state?}
   then deliver(res)
   else signal(failure);
end
```

The behaviour of SCOP can be described by the above control algorithm with comments on the right side. The `decide` procedure determines the maximum number `max_phase` of possible phases to be permitted by the specified timing constraints. Procedure `configure` constructs the CAS set V_1 in phase one according to the selected delivery condition and the given application environment, and establishes the CAS set V_i ($i>1$) based on the syndrome S_{i-1} collected in the $(i-1)$th phase and the information on phases. The execution of a CAS may lead to a successful state `E`. Note that variants in V_i are selected from the variants that have not been used in any of the previous phases, i.e. V_i is a subset of $V - (V_1 \cup V_2 \cup \ldots \cup V_{i-1})$. If the ith phase is the last, V_i would contain all the remaining spare variants. The `execute` procedure manages the execution of the variants in CAS and generates the syndrome S_i, where S_0 is an empty set and S_{i-1} is a subset of S_i. Procedure `adjudicate` implements the adjudication function using the selected condition `C`. It receives the syndrome S_i, sets the new `State_mark` and selects the result `res`, if one exists. The `deliver` procedure delivers the selected result and the `signal` produces a failure notification.

Example: Suppose that i) three processors are available for execution of up to three software variants; ii) seven software variants are provided; iii) the maximum time delay permitted is three phases, and iv) a result selected from at least three agreeing versions is considered as deliverable. An example of possible executions is illustrated in Table I, where italics are used to indicate the correct results, bold characters to represent the incorrect, disagreeing results, and $V=\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$.

Phase	V_i	Spares	Syndrome	Judgement & result
1	$\{v_1,v_2,v_3\}$	$\{v_4,v_5,v_6,v_7\}$	$\mathbf{r_1}$, r_2, $\mathbf{r_3}$	N
2	$\{v_4,v_5\}$	$\{v_6,v_7\}$	$\mathbf{r_1}$, r_2, $\mathbf{r_3}$, r_4, r_5	$\Rightarrow$ E, r_2

Table I Execution example of SCOP.

This example shows how SCOP reaches the required dependability in a dynamic manner when the availability of hardware resources is fixed. The amount of available resources and real-time constraints may vary in practice. Particularly, in computing systems that function for a long period of time, the user may impose new requirements or modify the original requirements for timeliness, dependability etc. Also the user may program more variants when the need arises. These uncertain factors require more complicated and more dynamic control and management. SCOP is further intended to handle this.

3.2 Dynamic Behaviour in a Multiprocessor Environment

In a large multiprocessor system, hardware resources can often be utilized by several competing concurrent applications. Furthermore, complex schemes for software fault tolerance may be necessary only for some critical part of the application that demands extra resources from the system. Dynamic management can make the allocation of resources more efficient. Let N_p be the maximum number of software variants which can be executed in parallel on hardware resources currently allocated to the given application, and T_d the time deadline that indicates the maximum response delay permitted for an application. Figure 2 illustrates a possible organization for SCOP and its dynamic behaviour in such a varying environment.

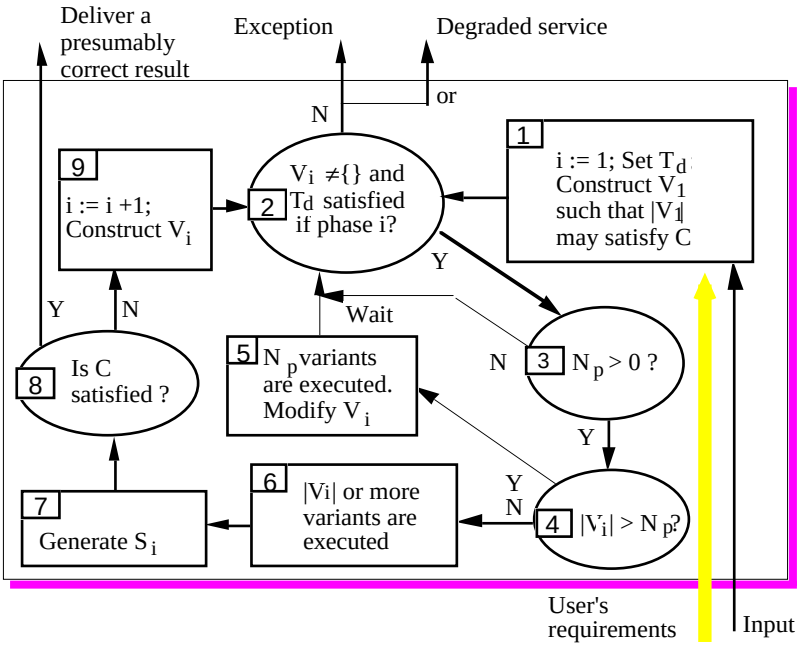

Fig. 2. Dynamic behaviour of SCOP in a varying environment.

Each step in the diagram is explained as follows.

1) For a given application, SCOP first establishes the delivery condition C (or several delivery conditions that permit different levels of dependability), according to the user's requirements, and then configures the CAS set, V_1, from V, that includes the minimum number of variants needed to be executed to generate a result satisfying the condition C in the absence of faults. The timing constraint, T_d, is also determined based on the response requirement.
2) Check whether the time deadline T_d will be missed when the ith phase is initiated for the execution of V_i (i = 1, 2, 3, ...) and whether V_i is an empty set. If T_d allows for no new phase or V_i ={ }, an exception will be raised to signal to the user that a timely result satisfying the required condition C cannot be provided. In this case, a degraded service may be considered.
3) Check whether $N_p > 0$. $N_p = 0$ means that no variant can be executed at this moment due to the limitation of available resources in the system. Wait and go back to Step 2 to check T_d.
4) Check whether $|V_i| > N_p$. If $|V_i| > N_p$, only some of the variants in V_i can be carried out within the current phase and thus additional time is needed for the execution of V_i.
5) N_p software variants in V_i are executed and V_i is modified so that V_i excludes the variants that have been executed. Back to Step 2.
6) Since $|V_i| \leq N_p$, $|V_i|$ variants are executed and completed within the current phase. If the scheduler used by the supporting system allocates $N_p > |V_i|$ processors to SCOP during the ith phase, it is possible to consider the execution

of N_p variants (more than $|V_i|$). This avoids wasting the resources that would be left idle otherwise, and requires the ability to select the additional variants among those not yet used.

7) Syndrome S_i is generated based on all the information collected up to this point.
8) Check whether a result exists that satisfies the delivery condition C. If so, deliver the result to the user; otherwise Step 9.
9) Set $i = i + 1$ and construct a new CAS set, V_i, from the spare variants according to the information about the syndrome, the deadline and the resources available; if no sufficient spare variants are available, set V_i empty.

It is worth mentioning that the major purpose of this illustration is to outline the dynamic behaviour of SCOP. Practical applications will require deliberate designs. What we need is a design methodology for supporting this kind of dynamic management and control. However, in many existing methodologies for programming dependable applications, the dependability aspects of an application are fixed statically [Agha, 1992 #1]. This is unsatisfactory for reasons of efficiency, flexibility and extendibility. We will propose in the next section a new methodology and show how (simple or complex) instances of SCOP could be derived automatically.

4 Design Methodology for SCOP

4.1 Description of a Design Methodology

In order to be able to adjust dependability and efficiency dynamically, the task of the control algorithm and of the adjudicator of SCOP may be very complex. This complexity could itself be a source of errors and thus defeat the proposed scheme. The development of a methodology for designing SCOP components is mainly meant to avoid such phenomena. Given the requirements to be fulfilled and proper information on the variants, an off-line activity is conducted for analyzing and collecting all the information regarding the possible (normal or abnormal) state of the software variants and the adjudicator. The correctness of the derived information can then be verified. Such off-line analysis and computation afford the major part of complexity of control and management, and could be made automatic by the construction and use of an appropriate tool. In this way, complexity of on-line control is greatly reduced. The adjudication mechanism and control algorithm of SCOP, which may be reusable for all SCOP instances, can take run-time actions by just identifying the states reached so far and by reading the associated information, without performing complex computation at each activation. This obviously improves dependability since the information collected in the off-line activity is made read-only and can be recorded on stable storage if necessary. The details of both off-line and on-line activities are illustrated in Figure 3.

Design Parameters: The design parameters of a SCOP component should include:

i) the number N of available variants, an estimate of reliability of each variant and an estimate of the execution time of each variant; (Estimates of the probability of common-mode failures among the variants are also desirable, but they are very rarely available in practice.)
ii) a timing constraint that the SCOP instance may be given for producing timely results; and

iii) the delivery condition(s) and the degree of flexibility in delivering results.

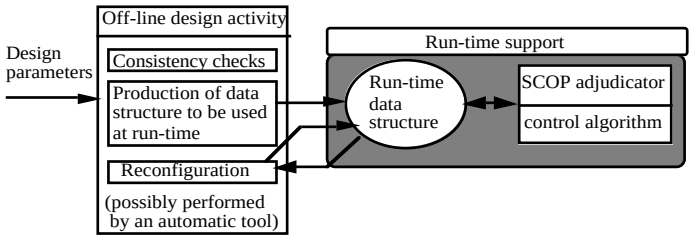

Fig. 3. Design and run-time organization of SCOP components.

In principle, a specific application has the freedom of establishing the delivery condition, whatever may satisfy the given dependability requirements. The degree of flexibility may range from only one fixed condition, which the SCOP component will try to verify for all the services, to many different conditions, one of which will be selected at run time by SCOP users through a run-time service parameter. Of course, permitting multiple delivery conditions will increase the amount of the off-line work necessary to provide the run-time information. Different delivery conditions may have respective fault coverages which may be difficult to compare. However, it is particularly useful to establish different delivery conditions whose coverages are totally ordered. A convincing example is the design of degradable systems: the same application may accept degraded services in the occurrence of faults. More generally, a given application may request different services under different conditions of the system state and its environment.

Off-line Design: The initial activity is, once the design parameters have been provided, to confirm their consistency: the parameters must admit solutions. For example if the response time required is shorter than the execution time of the variants, no timely services can be provided. It must also be verified that the design intended will be able to meet the delivery condition(s), probably using all the available redundancy when the need arises. Should the parameters prove to cause any inconsistency, they must be changed and the whole process started anew.

Next, a data structure, such as a directed graph or a table, is built. Each node or entry of the chosen structure will represent a possible state of execution of the scheme. It will contain, for each delivery condition, the corresponding state mark *E* (end-states), *N* (non end-states) or *F* (failure states). In case of the state *N* it will contain also the information on the necessary amount of redundancy to be further used. Each execution state corresponds to a set of syndromes that may lead to the same decisions under a given delivery condition. We will show later how such a graph may be actually built and discuss some minor differences when considering a table.

A further action is to define the initial state and to construct, for each delivery condition, the set of variants to start with, i.e. those to be executed in the first phase.

Run-Time Support: The run-time data structure is associated with the adjudicator and the control algorithm to support the SCOP run-time actions. Both the adjudicator and the control algorithm are independent of the data provided by the off-line design. At the end of each phase, the adjudication mechanism needs just to identify the syndrome collected (or the execution state) and then read from the data structure its classification

(*N*, *E* or *F*) with respect to the required delivery condition for that service. If the state is *N*, the control may dynamically configure a set of the variants for the next phase by looking up the information in the corresponding field of the data structure. The work at run-time is thus limited to following the information in the data structure, i.e. just moving from one element to another according to the observed syndrome.

Reconfiguration: The SCOP component will be operational waiting for service requests until affected by hard (permanent) faults. Although software components are usually modelled as being affected only by transient faults, the occurrence of hard faults should not be neglected. When hard faults need to be treated, a reconfiguration must be performed. The component will transit from an operational state to a reconfiguration state. For each variant that could be affected by a hard fault, the following off-line steps are performed to modify the data structure:

1) all the states representing execution of all variants (syndromes of *N* results) are erased from the data structure;
2) the faulty variant is extracted from the set of available ones; and
3) for each delivery condition, all the elements with the non end-states must be checked. If the reduced number of available variants no longer allows any end-state to be reached from a non end-state, then the state must be classified as a failure state.

This methodology consists essentially in providing the information off-line. Obviously algorithms for providing it may differ widely depending on the design parameters given. They could be made automatic and be grouped into a tool that helps use them appropriately. This automatic tool could then be used for designing SCOP instances and would be integrated into a programming environment for software development. Software designers could then use it to automatically derive instances of SCOP. They would just set the desired parameters without having to perform the analysis for any instance of SCOP they may want to design.

4.2 The Design of a SCOP Instance

We now describe, in order to show how the methodology can be applied, the off-line design process related to the following design parameters:

i) five software variants are available, their maximum execution time and figures of reliability are given, and all variants have the same reliability figures;
ii) there are no constraints on the response time for any service requests; and
iii) two different delivery conditions are defined: (1) C_1 — deliver a result if it is provably correct provided that no more than two variants produce incorrect results; and (2) C_2 — deliver a result if it is provably correct provided that no more than one variant produces incorrect results.

To check the consistency, we simply have to show that the condition C_1 can be met since C_1 has higher fault coverage than C_2 and no timing constraints are given. Note that some of the states reached by executing all the five variants can be marked as end-states, e.g. those representing at least three agreeing results. The requirement could also be met executing just three variants if all give agreeing results.

The second step consists in enumerating all the possible syndromes for each possible number M ($1 \leq M \leq 5$) of variants needed to be executed. Here, the frequency of a

result in a syndrome is defined as the times that the result appears in the syndrome. These syndromes are then partitioned into different classes, each of which is 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 frequency of the ith most frequent result in the syndrome. For example, when $M = 3$, the syndromes containing just two equal results belong to the class denoted as (2,1). Each class constitutes an execution state of the SCOP component we are designing, and is represented by a node in a directed graph.

In order to build such a directed graph we first define a relation among classes. The class S_i^M (ith class while executing M variants) is said to be related to the class S_j^{M+1} (jth class while executing M+1 variants) if it is possible to obtain from any syndrome in S_i^M, when a further variant is executed, a syndrome belonging to S_j^{M+1}. For example, from any syndromes belonging to the class (2,1), the execution of a fourth variant may lead to a syndrome that belongs to (3,1), and the class (2,1) is thus related to the class (3,1). A directed graph can be thus created — nodes represent the states (or the syndrome classes) and directed arcs represent the defined relation among classes. In Figure 4 the resulting graph is depicted. This directed graph structure has a nice property that one node is linked only to the states which may be reached by continuing execution; so any execution of the scheme can be monitored by following a path in the graph. A table structure where the strict linkage among reachable states is not directly represented would require a larger run-time effort for seeking the current state.

Next we will show how to derive the information to be associated with each node in the graph. For each node S corresponding to the execution of M variants (labelled with an ordered string of j numbers (z_1, z_2, ..., z_j) that indicates the result frequency), let $F_i(S) = M - z_i$ for i = 1, 2, ..., j, and $F_{j+1}(S) = M$. Note that $F_1(S)$ is the minimum number of variants that have failed and $F_{j+1}(S)$ represents the failure of all the M variants. More generally, $F_i(S)$ is equivalent to the number of the variants that have failed when the ith result is correct. It follows that $F_1(S) \leq F_2(S) \leq ... \leq F_j(S) < F_{j+1}(S)$. Let f_i be the maximum number of faulty variants admitted by the delivery condition C_i. We can label the nodes with respective states now.

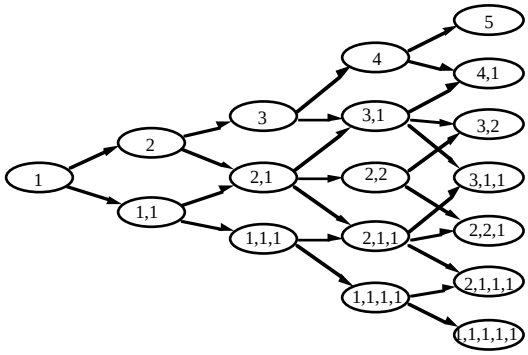

Fig. 4. Syndromes graph for N = 5.

For a given node, if $f_i < F_1(S)$, more than f_i failures must have occurred. The node should be associated with the failure state. If $F_1(S) < F_2(S)$ and $f_i < F_2(S)$, then a

most frequent result exists and the result can be unambiguously identified as correct. Otherwise, $F_1(S) = F_2(S)$ or $f_i \geq F_2(S)$ implies the result that appears z_2 times might be correct, whereas the result that appears z_1 times might be incorrect. Thus, the node should be labelled the non end-state. For example, the node (3) in Figure 4 represents three agreeing results out of the execution of three variants. For the node, F_1(node 3) = 0 and F_2(node 3) = 3. The result can be unambiguously identified as correct if it is assumed that at most two variants can have failed (i.e. f_i = 2 and F_1(node 3) < f_i < F_2(node 3)). The node (3) can be therefore associated with the end-state. Formally, for a given value f_i (induced from the condition C_i), a node S is labelled: i) end-state E if $F_1(S) \leq f_i < F_2(S)$; ii) non end-state N if $F_2(S) \leq f_i$; or iii) failure state F if $F_1(S) > f_i$.

Unlike the nodes with the state E or F, a node with the state N should contain extra information on the minimum number x of the variants to be further executed in the next phase so as to possibly reach an end-state. Now let $S = (z_1, z_2, ..., z_j)$ be a non end-state, where $F_2(S) \leq f_i$. Consider the best case in which the execution of the x variants produces the results that agree with the most frequent result in the state S. A new state $S^* = (z_1+x, z_2,..., z_j)$ will be reached. Note that $F_1(S^*) = M + x - (z_1 + x) = M - z_1 = F_1(S) \leq f_i$, and $F_2(S^*) = M + x - (z_2) = F_2(S) + x$. By the rule of labelling, the state S^* is an end-state if $f_i < F_2(S^*)$. Since $f_i < F_2(S^*)$ means $f_i < F_2(S) + x$, we have that $x > f_i - F_2(S)$, or $x = f_i + 1 - F_2(S)$.

The last step we have to take is to define the initial state and to set the number of variants to be executed in the first phase for the delivery conditions $\{C_i\}$. The initial state is defined as a null state with $F_1(0) = F_2(0) = 0$. Since it is a non-end state, the rule just described above can be applied and the initial number of variants to be executed is: $x = f_i + 1 - F_2(0) = f_i + 1$.

Table II summarizes all the data obtained at the end of the off-line activity. There are nineteen syndrome classes in this instance. Each class is associated with a state mark under the specific delivery condition. If the state mark is N, the corresponding x shows the minimum number of variants needed to be executed in the next phase. (•• is used for cells with no information.) Note that the off-line activity also helps identify different delivery conditions. For example, consider a condition, C^*, that a result is deliverable if it is selected from three or more agreeing results. This condition seems to be the same as the condition C_1 used in the previous SCOP instance. However, they are actually different. The state (2, 1, 1) in Table II is an end-state under the condition C_1, but a non end-state according to the condition C^*.

	C_1 : [0, 2]		C_2 : [0, 1]			C_1 : [0, 2]		C_2 : [0, 1]	
Class	Mark	*x*	Mark	*x*	Class	Mark	*x*	Mark	*x*
5	E	••	E	••	2,1,1	E	••	F	••
4,1	E	••	E	••	1,1,1,1	F	••	F	••
3,2	E	••	F	••	3	E	••	E	••
3,1,1	E	••	F	••	2,1	N	1	E	••
2,2,1	F	••	F	••	1,1,1	N	1	F	••
2,1,1,1	F	••	F	••	2	N	1	E	••
1,1,1,1,1	F	••	F	••	1,1	N	2	N	1
4	E	••	E	••	1	N	2	N	1
3,1	E	••	E	••	0	N	3	N	2
2,2	N	1	F	••					

Table II Off-line information obtained for the instance of SCOP with five variants.

5 Evaluation

5.1 Dependability Aspects

We first analyse the SCOP scheme based on a configuration that makes it comparable with the other major schemes. Arlat *et al.* [Arlat, 1995, chapter 6.5, internal reference] developed complete fault classifications and constructed fault-manifestation models for RB, NVP and NSCP. We shall exploit their framework for modelling, with some slight adjustments, and use some of their results for the purpose of our comparison. Four schemes are considered: i) the SCOP architecture involves three variants, the delivery condition associated with it requires that at least two variants produce the same results, and it thus executes just two variants in the first phase; ii) the NVP architecture uses three variants based on the usual majority adjudication; iii) the RB architecture consists of a primary block and an alternate block; and iv) NSCP contains four variants organized as two self-checking components.

In our fault-manifestation model, two or more design faults among different components (e.g. variants and the adjudicator) are considered as "related" if they cause similar or identical errors; and they are regarded as "independent" if the errors they cause are distinct. In accordance with the consideration of general applicability, probabilities of independent and related faults are allowed here to be significantly high, with the upper bound up to one. The following analysis only concerns value correctness and does not deal with timing constraints. We focus here on reliability and safety as two attributes of dependability, but further extension could be made including other measures such as availability and performability [Tai, 1993 #10].

Basic Assumptions and Notation: Let *X* indicate one of the four schemes to be considered: SCOP, NVP, RB, NSCP. The analysis will be made using the following assumptions.

i) Fault types and notation for probabilities of fault manifestation:

a) independent fault(s) in the adjudicator or in the variant(s), given no a manifestation of related faults — the fault in the adjudicator manifests with probability $q_{A,X}$ while the fault in a variant with probability q_{IX};

b) a related fault between the variants and the adjudicator that manifests with probability $q_{VA,X}$;

c) a related fault among S variants that manifests with probability $q_{SV,X}$.

We condition the probability $q_{A,X}$ on a conservative base, namely, the fault will always cause the adjudicator to reject a result (or a majority) given the result (or the majority) is correct, or the adjudicator to output a result given the result is incorrect and no majority exists. Note that a comparison-based adjudicator is normally application-independent. We will not consider the fault type "b)" in the models for SCOP, NVP and NSCP, but in the model for RB.

ii) The probability of a manifestation of an independent fault is the same for all variants.

iii) During the execution of Scheme X, only a single fault type may appear and no compensation may occur between errors of the variants and of the adjudicator.

iv) For the reason of simplicity, it is assumed that an independent fault in the adjudicator will only manifest itself at the end of the final adjudication.

We can now compute the probability of software failure based on a discrete-time, finite-state Markov process. Due to the limitation of space, we have to omit all details of our dependability models. The interested reader is referred to [Di Giandomenico, 1993 #29]. For the X approach, let $Q_{R,X}$ be the probability of software failure and $Q_{S,X}$ be the probability of undetected failure, we obtain that

$$Q_{R,SCOP} = (q_2 - 2q_Aq_2 + q_Iq_1 - 2q_Aq_Iq_1 + q_A)(1 - q_3 - q_4) + Q_{S,SCOP}$$
$$Q_{S,SCOP} = q_A(q_Iq_1 + q_2)(1 - q_3 - q_4) + q_3 + q_4$$

where $q_1 = 2q_I(1-q_I)$, $q_2 = q_I^2$, $q_3 = 2q_{2V}$, and $q_4 = q_{2V} + q_{3V}$.

For the purpose of comparison with the NVP architecture, let $q_i = 3q_I^2(1-q_I) + q_I^3$ and $q_r = 3q_{2V} + q_{3V}$. It follows that

$$Q_{R,SCOP} = q_A(1-q_i)(1-q_r) + (1-q_A)q_i(1-q_r) + Q_{S,SCOP}$$
$$Q_{S,SCOP} = q_Aq_i(1-q_r) + q_r$$

Following a similar approach to dependability modelling, we conclude that

$$Q_{R,NVP} = q_A(1-q_i)(1-q_r) + (1-q_A)q_i(1-q_r) + Q_{S,NVP}$$
$$Q_{S,NVP} = q_Aq_i(1-q_r) + q_r$$
$$Q_{R,RB} = q_A(1 - q_I - q_{2V} - q_{VA}) + q_Aq_I(1-q_I) + (1-q_A)q_I^2 + q_{2V} + Q_{S,RB}$$
$$Q_{S,RB} = q_Aq_I^2 + q_{VA}$$
$$Q_{R,NSCP} = q_A(1-q_{iv})(1-q_{rv}) + (1-q_A)q_{iv}^2(1-q_{rv}) + 4q_{2V} + Q_{S,NSCP}$$
$$Q_{S,NSCP} = q_Aq_{iv}^2(1-q_r) + q_{rv} - 5q_{2V}$$

where $q_{iv} = 2q_I(1-q_I) + q_I^2$ and $q_{rv} = 6q_{2V} + 4q_{3V} + q_{4V}$.

From these specific expressions for benign and catastrophic failures, we could claim that SCOP has the same level of dependability as NVP. However, the adjudicator in the SCOP architecture under consideration will be utilized twice if some faults are detected in the first phase. It is therefore possible that SCOP delivers an incorrect result in the early phase or rejects a correct majority, starting a new phase, because of a manifestation of a fault in the adjudicator. When such fault situations are taken into account, the dependability of SCOP would be slightly lower than that of NVP with a

simple majority voter. In fact, probabilities associated with the adjudicator of NVP may also vary significantly since various complicated adjudicators may be employed [Di Giandomenico, 1990 #30].

RB seems to be the best, but an AT is usually application-dependent. The degree of design diversity between an AT and the variants could be different with respect to different applications so that $q_{VA,RB}$ (i.e. the probability of a related fault between an AT and the variants) may vary dramatically. Besides, the fault coverage of an AT is an indicator of its complexity, where an increase in fault coverage generally requires a more complicated implementation. NSCP could suffer from related faults among variants in spite of its low $q_{A,NSCP}$ (simple adjudication based on result comparison and switch).

5.2 Consumption of Resources

Let T_X be the time necessary for the execution of a complete phase in the X scheme, where T_X consists of the execution time of the variants and the time for adjudication and control, and f be the maximum number of variant failures to be tolerated by the scheme. We now conduct an analysis of resource consumption, referring to more general architectures (than those used in the dependability analysis): i) the SCOP architecture involves $2f$+1 variants, with the delivery condition that requires at least f+1 identical results, executing f+1 variants in the first phase; ii) the NVP architecture uses $2f$+1 variants based on the usual majority adjudication; iii) the RB architecture consists of a primary block and f alternate blocks; and iv) NSCP contains 2(f+1) variants organized as (f+1) self-checking components. (For NSCP, it must be further assumed that all the variant failures are independent.)

In the interest of concentrating on efficiency, we assume no timing constraints for the service and perfect adjudicators (and controllers). Table III reports some results about resource consumption, in which each cell of the NoVariant column indicates the total number of variants needed to be executed while a scheme attempts to complete its service, in direct proportion to the amount of hardware resources. Both the worst and the average case are shown. In the worst case, SCOP requires the amount of hardware resources that supports the execution of ($2f$+1) variants, but on an average it needs the hardware support just for the execution of the (f+1) variants. It would be therefore reasonable to rank SCOP as being more efficient than NVP and NSCP. RB would seem to be better again, relying to some extent on the use of acceptance tests.

SCOP will terminate any further execution if (f+1) agreeing results are obtained in the first phase, i.e. the condition for delivering a result is satisfied. This scenario happens when i) a related fault manifests itself and affects all the (f+1) variants; or ii) all the variants produce the same correct result. Events like the failure or success of individual variants are usually not independent but positively correlated under the condition that all the variants are executed together on the same input [Knight, 1986 #31]. This factor determines that the probability of observing the event "ii)" would be higher than what might be expected assuming independence. Let p_V be the probability that a single variant gives a correct result. The probability that SCOP would then stop at the end of the first phase > the probability that the f+1 variants would produce correct results > p_V^{f+1}. Experimental values of p_V [Knight, 1986 #31] are sufficiently high so that we would claim SCOP almost always gives the same fast response as NVP or even faster, as discussed later. Note that the worst case in which SCOP

operates with the longest execution time $T_{SCOP}+fT_{SCOP}$ has in fact a very rare probability of occurrence. It occurs only when the first phase ends with f agreeing results and every remaining variant, assigned to run in one of phases Two to (f+1), produces a different result.

Scheme	NoVariant worst	NoVariant average	TIME worst	TIME average
SCOP	(f+1)+f	≅(f+1)	$T_{SCOP}+fT_{SCOP}$	$\cong T_{SCOP}$
NVP	2f+1	2f+1	T_{NVP}	T_{NVP}
RB	1+f	≅1	$T_{RB}+fT_{RB}$	$\cong T_{RB}$
NSCP	2(f+1)	2(f+1)	$T_N+ft_{switch}^{\dagger}$	$\cong T_N$

† t_{switch} is the time for switching the self-checking components

Table III Comparison of resource consumption.

If timing constraints for delivering the result are considered, for example given that the maximum number of allowable phases is p where $p \leq f$+1, the SCOP's worst case of execution time will become pT_{SCOP}. Note, however, that this limitation on the number of phases heavily impacts the average usage of variants only if $p = 1$ (in this case the execution of all the variants is required). Otherwise the first phase, very likely the only one, always involves just (f+1) variants. The basic RB scheme will not be applicable directly to the case that $p < f$+1, as RB needs f+1 phases to deal with successive manifestations of f faults. Parallel implementations of RB may be suitable, but they operate at the cost of more variants executed within a phase.

More precisely, the execution times of various adjudication functions (and control algorithms) can be significantly different with respect to specific fault coverages and algorithm complexity. Furthermore, the execution times of variants in a scheme can differ because of the requirements for design diversity and equivalent variant functionality. Since in SCOP the subsequent phases will be utilized much less than the first phase, the faster variants (those that correspond to more effective implementations and whose execution times are shorter) may be chosen in the first phase. The penalty caused by variant synchronization (that requires the system to wait for the slowest variant) can be thus reduced, as compared with NVP.

Example: (*Comparison of SCOP and NVP*) Table IV gives the related figures of resource consumption in SCOP and NVP where $T_{SCOP} \leq T_{NVP}$ (i.e. $T_{SCOP1} \leq T_{3VP}$, $T_{SCOP2} \leq T_{5VP}$, and $T_{SCOP3} \leq T_{7VP}$) and $p_V = (1 - 10^{-4})$ (the average reliability of the variants derived from the experiment in [Knight, 1986 #31]). Data for SCOP have been obtained based on the assumption that just two phases are allowed. The table shows that SCOP consumes almost the same amount of time as NVP to provide services, but it requires just the amount of hardware resources that supports (f+1)VP (which only resists at most f/2 variant failures), rather than (2f+1)VP.

	Scheme	No. of variants executed (Average)	Time consumption (Average)
Average Cost for General Case	SCOP	$(f+1)+(1-p_V^{f+1})f$	$[1+(1-p_V^{f+1})]T_{SCOP}$
	NVP	$2f + 1$	T_{NVP}
N = 3 f = 1	SCOP	2.0002	$1.0002T_{SCOP1}$
	NVP	3	T_{3VP}
N = 5 f = 2	SCOP	3.0006	$1.0003T_{SCOP2}$
	NVP	5	T_{5VP}
N = 7 f = 3	SCOP	4.0012	$1.0004T_{SCOP3}$
	NVP	7	T_{7VP}

Table IV Resource consumption of SCOP and NVP.

6 Conclusions

In this paper we have discussed in detail the problem of attaining a flexible compromise between using redundancy to improve dependability and limiting the amount of redundancy for the purpose of efficiency. The SCOP approach has been developed with the aim of dynamically adjusting different characteristics that matter in fault-tolerant software. The results drawn from the evaluation of dependability show that SCOP could have the same level of dependability as other existing schemes. To attack the problem of complexity caused by highly dynamic behaviour, we have developed a methodology for devising various instances of SCOP, simplifying the on-line process at the price of the complex (but systematic and thorough) off-line design.

SCOP gives designers a general control framework into which multiple version software, diversity in data space, or simple copies of a program can be incorporated selectively, depending upon considerations of cost effectiveness and specific application requirements. The SCOP concept also admits different delivery conditions for the results produced. Since the different conditions may have different coverages SCOP could become parametric to dependability. This represents a significant novelty in software fault tolerance. SCOP makes it possible to impose additional dependability requirements for a system (or a server) or to provide gracefully degraded services flexibly as faults occur. What is to be sacrificed can be dynamically decided at run time with respect to the resources that are currently available.

In principle, SCOP provides the same basis as NVP and RB for achieving software fault tolerance and has no major implementation difficulties in any environments where NVP or RB has found considerable applications. A few existing applications have provided us with additional confidence for adopting SCOP in practical systems. For example, four-version software and two-variant hardware are combined in a *dynamically reconfigurable* architecture to support the pitch control of the *fly-by-wire* A320 aircraft [Traverse, 1988 #32], which could be regarded as a limited or simplified form of the SCOP technique.