The $t/(n-1)$-VP Approach to Fault-Tolerant Software

Jie Xu and Brian Randell

Department of Computing Science

University of Newcastle upon Tyne, UK

ABSTRACT

This paper describes a software fault tolerance scheme, called $t/(n-1)$-Variant Programming (or $t/(n-1)$-VP), which is based on a particular system diagnosis technique used in hardware and thereby has some special advantages involving a simplified adjudication mechanism and enhanced capability of tolerating faults. A detailed dependability evaluation of the $t/(n-1)$-VP architecture is conducted, compared with two similar schemes, namely, N-version programming (NVP) and N self-checking programming (NSCP). The results drawn from the comparison clearly show that $t/(n-1)$-VP is a viable addition or alternative to present techniques.

Much of classical work on dependability analysis of software fault tolerance approaches has focused on the simplest architectural examples that can only tolerate single software faults, without considering tolerance to multiple and related faults. The results obtained from such analyses are thus restricted. The dependability evaluation carried out in this paper deals with more complicated and general software redundancy, *ie*, various architectures tolerating two or more faults. It is not a great surprise that we come to new conclusions: both $t/(n-1)$-VP and the NVP scheme have the ability to tolerate some related faults between software variants; in general, $t/(n-1)$-VP has higher reliability, whereas NVP would be better from the safety point of view.

Key words — Related software faults, Software fault tolerance, Software reliability, Software safety, System-level fault diagnosis

To appear in *IEEE Trans. on Reliability*, vol.46, no.1, March 1997.

Acronyms

NMR	*N*-modular redundancy
NSCP	*N* self-checking programming
NVP	*N*-version programming
NVS	Sequential application of NVP
RB	Recovery blocks
$t/(n-1)$-VP	$t/(n-1)$-variant programming

1. INTRODUCTION

The provision of tolerance to anticipated hardware faults has been a common practice for many years and forms a vital part of any dependable computing system. A relatively new development is the fault tolerance techniques for coping with unanticipated faults such as design (typically software) faults [1]. In principle, simple replication of software components is insufficient because software design faults can be reproduced when only redundant copies are made [2]. Software fault tolerance usually requires the application of *design diversity*. Design diversity is the approach in which two or more *variants* of a component for redundant computations are independently designed to meet a common service specification. Variants are aimed at delivering the same service, but implemented in different ways in the hope that they do not contain the same design faults. Since at least two variants are involved, tolerance to design faults necessitates an *adjudicator* [3] (*ie*, a decision algorithm) that determines a single (assumed to be) error-free result based on the results produced by multiple variants. Several techniques have been proposed for structuring a software system, and providing software fault tolerance: *recovery blocks* (RB) [2], *N-version programming* (NVP) [4], *N self-checking programming* (NSCP) [5] and some intermediate or combined techniques (see [6, 7] for example). These techniques should be regarded as complementary to (not as a substitute for) those for achieving software fault avoidance such as verification & validation, software testing, and proof methodology.

The first scheme developed for achieving software fault tolerance was the recovery block scheme [2, 8], in which variants are organized in a manner similar to the standby sparing technique [9] used in hardware. The recovery block approach performs run-time fault detection by augmenting any conventional hardware/software error detection mechanism with an acceptance test applied to the results of execution of one variant. If the test fails, an alternate variant is invoked after backward error recovery is performed. Researchers at UCLA devised another approach, namely the *N*-version programming scheme [4]. The NVP approach is a direct application of the hardware *N*-modular redundancy approach (NMR) [9] to software. *N* versions (*ie*, variants) of a program that have been independently designed are executed in parallel and their results compared by an adjudicator. By incorporating a majority vote, the system can eliminate erroneous results (*ie*, the minority) and pass on the (presumed to be correct) results (*ie*, the majority). In simple cases the voting can be based on tests for identicality; in general, a more sophisticated and application-oriented test is needed. Note that *N* variants in this approach may be executed sequentially. Grnarov, Arlat and Avizienis [10] sketched such a sequential application of NVP, called NVS. Laprie *et al.* [5, 11] in 1987 identified a new scheme: *N* self-checking programming. The NSCP approach attains fault tolerance by the parallel execution of *N* self-checking software components. Each self-checking component consists of a pair of variants with a comparator. In particular, one self-checking component is regarded as the active component, and the others are considered as “hot” stand-by spares.

It must be recognized that the success of a fault tolerance scheme depends to a great extent upon its adjudicator and unreliability in the adjudicator can have a dramatic impact on the overall system reliability [1]. The design for a highly reliable adjudicator generally requires that 1) the adjudication mechanism and variants being checked are as independent as possible, so that they cannot be affected by common faults or *related faults*; 2) the mechanism itself must be simple enough to guarantee its reliability and the overall system performance. The traditional mechanisms are not entirely satisfactory. In the recovery block software, an acceptance test is used in its adjudication mechanism to provide a last line of detecting errors, but since the test is system-specific, and as such very little specific guidance can be given for its construction, it is difficult to ensure that the acceptance test and variants will be independent of each other. To overcome this problem, some schemes adopt an adjudication mechanism that selects the results by comparing the outputs of multiple variants. However, a practical adjudicator used in NVP is much more sophisticated than the early simple majority vote, while adjudication mechanisms constructed in NSCP are somewhat too simple to effectively detect the related faults that may occur in the active self-checking components. We develop an alternative in the next section, called $t/(n-1)$-*Variant Programming* ($t/(n-1)$-VP), which exploits several new research results in the area of system diagnosis [12, 13] for the design of a simplified adjudication mechanism. Our proposed scheme has several favourable characteristics, including 1) the potential ability to tolerate multiple related faults among variants, 2) simple adjudication mechanism that requires only $O(n)$ result comparison steps, 3) the delivery of correct service even when the number of faulty variants exceeds the bound t in some fault situations, and 4) possible forms of graceful degradation.

Note also that it cannot be guaranteed that independently designed variants will fail independently (*ie*, that faults in the different variants will occur at random and be unrelated) despite the adoption of the design diversity approaches [14, 15]. The dependability analysis of software-fault-tolerant systems must therefore study the effect of related (or dependent) faults. A number of papers devoted to such dependability analysis have appeared in the literature (see [6, 14 - 17] for example). In particular, Arlat, Kanoun, and Laprie [16] developed complete fault classifications and presented a detailed evaluation of NVP and RB. Their analysis concentrated on basic architectures able to tolerate a single fault and thereby the analytical conclusions can hold only for those specific instances. In the third section, we augment published work by analyzing more complex (more general) architectures that tolerate two or more software faults and by carefully identifying the ability of various approaches to tolerate independent and related faults. The results drawn from our analysis provide designers with richer information about the fault tolerance properties of various architectures than the results from traditional analysis, and show evidences that the $t/(n-1)$-VP approach is a viable addition or alternative to present schemes for coping with software faults.

Notation

A_i	state of adjudicator's execution
B	state of benign failure caused by an undetected error
C	state of catastrophic failure caused by an undetected error
C_i	result comparator
C_X	probability of catastrophic failure of the X approach
D	state of detected failure
E	state of software execution
F_X	probability of failure of the X approach
I	state of software idleness during the specified exposure period
N, n	number of software variants
p	probability that all variants produce the same correct results
q_I	probability of an independent fault in a variant
q_A	probability of an independent fault in the adjudicator
q_{AD}	probability of a detected independent fault in the adjudicator
q_{AU}	probability of an undetected independent fault in the adjudicator
q_{AV}	probability of related faults among the variants and the adjudicator
q_{mV}	probability of related faults among m variants
q_U	probability of an undetected failure
q_C	probability of a catastrophic failure due to an undetected error
$R_X(t)$	reliability of the X approach
$S_X(t)$	safety of the X approach
U	state of undetected failure
V	state of variant execution
V_i	ith software variant
σ	departure rate from state I
$\omega_{i,i+1}$	(comparison) test outcome

Other, standard notation is given in “Information for Readers & Authors” at the rear of each issue. Less frequently used notation may be defined in the text where it first appears.

2. THE $t/(n–1)$-VARIANT PROGRAMMING SCHEME

In the theory of system-level fault diagnosis (see [12, 13] where further references can be found), a particular diagnosability measure, denoted as $t/(n–1)$-diagnosability, was first introduced in [18]. Its diagnosis goal is, for a system composed of n units, to isolate the faulty units to a set of size at most $(n–1)$, under the condition that the number of faulty units is at most t. In other words, at least one unit exists such that it is not in the set of size $(n–1)$ and can thus be unambiguously identified as fault-free, provided that the system itself is $t/(n–1)$-fault diagnosable and the number of faulty units in the system does not exceed the bound t. Immediately, a natural idea springs to mind: the $t/(n–1)$-diagnosis technique may be employed to select a single correct result from the results generated by n replicated software modules (of independent design). An intuition indicates that we would benefit from the utilization of $t/(n–1)$-diagnosis since this special diagnosis measure cuts down significantly the requirement on the number of tests (*ie*, the number of result comparisons) relative to previous diagnosis schemes. It is thus possible to somehow use the idea behind the $t/(n–1)$-diagnosis technique to construct a simple, but dependable adjudication mechanism. Based on current theoretical results of $t/(n–1)$-diagnosis (see [19 - 24] and a subsequent discussion), we develop a new scheme for tolerating hardware and/or software faults. Our description of this scheme is first in terms of application to software fault tolerance, but the approach can also be implemented with hardware [22]. Two classes of software faults are distinguished: independent faults and related faults [11, 16]. Independent faults occur in single variants or in the adjudication mechanism, while related faults can take place among multiple variants and among the adjudicator and one or more variants.

2.1. Description of the $t/(n–1)$-VP Scheme and an Example

In what follows, we shall term such a software fault tolerance scheme ***t*/(*n*–1)-Variant Programming**. A general $t/(n–1)$-VP architecture can identify the correct result from a subset of the results of n software modules (or variants), provided that the number of faulty modules in the architecture does not exceed t (*ie*, it can tolerate at least t software faults). The semantics of $t/(n–1)$-VP can be expressed more directly as follows:

1) each of n independently designed software variants is executed in parallel;
2) just some of their results are compared to produce a syndrome;
3) using the syndrome, a diagnosis program performs $t/(n–1)$-diagnosis and selects a presumably correct result as the system output (*eg*, through switching of the results); and if no acceptable result is identified, the system will invoke spare software variants, if exists some, or simply signal an exception.

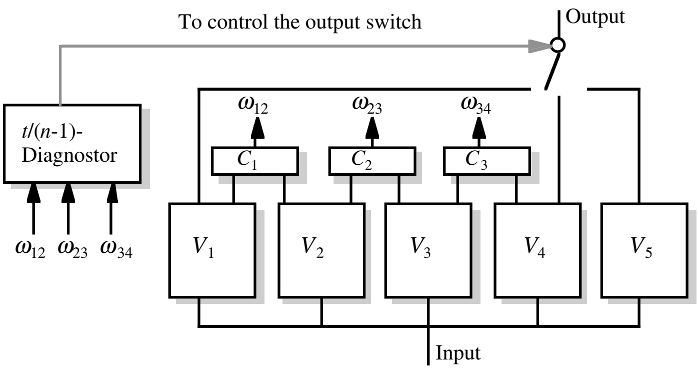

Figure 1. A $t/(n–1)$-VP architecture with $n = 5$ and $t = 2$.

For the proposed scheme, we first use a concrete example to demonstrate its ability to tolerate both independent and related faults, and then address its effectiveness for any given n and t. Figure 1 shows a $t/(n–1)$-VP architecture where $n = 5$ and $t = 2$. This 2/(5–1)-VP architecture consists of five independently designed software modules, called variants V_1, V_2, ..., and V_5, which are executed in parallel in a framework that is intended to cater for up to two simultaneous software faults. Three comparators C_1, C_2, and C_3 are placed at the outputs of variants V_1, V_2, V_3, and V_4 to perform error detection, where C_i compares the results of V_i and V_{i+1} ($i = 1, 2, 3$) and generates the test outcome $\omega_{i,i+1}$. Three (comparison) test outcomes ω_{12}, ω_{23} and ω_{34} constitute a syndrome. In particular, the test outcome $\omega_{ij} = 0$ (1) if the results of the variants V_i and V_j agree (disagree). A diagnosis program, the $t/(n–1)$-diagnostor, selects one of the results of V_1, V_4, and V_5 according to the value of the syndrome, and switches service delivery (*ie*, the system output) to the selected result. The adjudicator of the architecture is implemented by the three comparators, the $t/(n\text{–}1)$-diagnostor and the output switch. Note that the result "lines" of V_2 and V_3 are not connected to the output switch and V_5 is not connected to a comparator. However, we can show below that this architecture is $t/(n\text{–}1)$-diagnosable for $t = 2$; namely, the diagnostor can always select a correct result *provided* that the number of (independent or related) faults in variants does not exceed two.

Let r_1, r_2, ..., and r_5 be the results of variants V_1, V_2, ..., and V_5 respectively. Table 1 gives all possible syndromes and the corresponding results that can be unambiguously diagnosed as correct while assuming that no more than two faults occur simultaneously. For example, in the case that $\omega_{12} = 0$, $\omega_{23} = 1$ and $\omega_{34} = 0$, a single correct result cannot be simply identified from among those produced by variants V_1, V_2, V_3, and V_4. We can however infer from the syndrome that two or more of the variants V_1, V_2, V_3 and V_4 have generated incorrect results because one single fault cannot lead to such a syndrome. Hence the result of V_5 must be "correct". In the case where $\omega_{12} = \omega_{23} = \omega_{34} = 0$, either all of the variants V_1, V_2, V_3 and V_4 have to be correct or all of them have to be incorrect. By the previous assumption that $t = 2$, these results should be classified as acceptable. Following a similar method, we can analyze other cases to determine the correct results. In fact, table 1 may be viewed as a simple diagnosis algorithm for the specific architecture. From this table, we find a significant fact that at least one of results r_1, r_4 and r_5 must be correct for a given syndrome. Accordingly, this architecture can deliver the correct system output by choosing just among the results of three variants V_1, V_4 and V_5.

Table 1. Possible syndromes and result selections for the $t(=2)/(n\text{–}1=4)$-VP architecture.

ω_{12}	ω_{23}	ω_{34}	Presumably Correct Results
0	0	0	r_1 r_2 r_3 r_4
0	0	1	r_1 r_2 r_3
0	1	0	r_5
0	1	1	r_1 r_2
1	0	0	r_2 r_3 r_4
1	0	1	r_5
1	1	0	r_3 r_4
1	1	1	r_5

Unlike the NVP scheme and its variations, $t/(n–1)$-VP does not have to make pairwise comparisons among the results of n variants in order to identify a presumably correct result. It is however interesting to study how many result comparisons (corresponding to the comparators illustrated in Figure 1) are normally required for a general $t/(n–1)$-architecture. In the simplest case that $n = 3$ and $t = 1$, one comparator is necessary and sufficient for $t/(n–1)$-diagnosis — the third result must be acceptable when the two compared results disagree; otherwise they can be identified as correct. Larger n and t require a more deliberate comparison assignment among the results of multiple variants so as to guarantee $t/(n–1)$-diagnosability. For example, result comparisons of n variants could be organized into a form of chains, where the comparator C_i ($1 \le i \le n–1$) compares the results of variants V_i and V_{i+1}. Alternatively, result comparisons may be organized into a more complex structure, $H_{2r,n}$, where the result of variant V_i ($1 \le i \le n$) is compared with that of V_j if and only if $i–r \le j \le i+r$ (mod $n+1$, $r = 1, 2, 3, ...$). Theorem 4 in [22] gave several sufficient conditions on the result comparison assignments of the systems that are $t/(n–1)$-diagnosable, and it is reproduced here in order to make this paper self-contained.

Theorem: A system S composed of n units (or software variants) is $t/(n–1)$-diagnosable if $n \ge 2t + 1$ and the assignment of result comparisons in the system S contains at least

1) a chain of $t + 1$ units for $t = 1$;
2) a chain of $t + 2$ units for $t = 2$;
3) a chain of $2t +1$ units for $3 \le t < 5$;
4) an $H_{2r,n}$ structure with $r = 1$ for $5 \le t < 7$;
5) an $H_{2r,n}$ structure with $r \ge (t–1)/5$ for $7 \le t$.

Because the major aim of this paper is to show how the $t/(n–1)$-diagnosis technique could be applied to the design of software fault tolerance schemes, we will not further discuss this particular technique itself (the interested reader is referred to [20, 22 - 24] for more technical details). The diagnosis algorithms with respect to the testing assignments in the above theorem have been developed in [23] for chains and in [24] for $H_{2r,n}$-type systems. For practical values of n (*eg*, $3 \le n \le 10$), a $t/(n–1)$-VP architecture uses only $O(n)$ comparators and contains a simple diagnosis algorithm with linear complexity. The adjudicator in such an architecture would be simpler than a voter used in NVP (which has to be based on $O(n^2)$ result comparison steps).

It is important to realize that for any fault tolerance scheme the correctness of results output by the system cannot always be guaranteed (*eg*, when more than t faults have occurred), and moreover such fault situations cannot be detected completely. However this does not present severe problems; there are acceptable probabilities of catastrophic events in practice (*eg*, an aircraft computer system usually accepts the probability of failure that is less than 10^{-9} per hour in a ten hour flight [25]). Dependability studies that have been performed for practical fault-tolerant systems can be used to determine the probability of the occurrence of t faults. This helps to make an appropriate design decision as to which scheme is likely to be most effective and how many variants are sufficient for a particular application. Additional fault detection and exception handling techniques [26] can also be used to improve fault coverage and fault tolerance. In the 2/(5–1)-architecture of Figure 1, for example, exception-handlers can be incorporated into the variants. The function of the handlers in a variant is to handle any errors that are detected during the execution of the variant, signalling an exception to the diagnostor. The diagnostor comes to final decision according to the value of the syndrome and the exception signals received so far: either it delivers a presumably correct result, or signals a failure exception.

2.2. Comparison with Other Schemes

The $t/(n–1)$-VP scheme has resemblance with other fault tolerance techniques that have been previously proposed and examined, especially with those requiring the use of result comparisons such as NVP and NSCP. The fact that the variants are executed in parallel necessitates an input consistency mechanism and a synchronization regime, based essentially on wait and send primitives, and incorporating a time-out mechanism. However, in each case there are significant and fundamental distinctions. Correct results in $t/(n–1)$-VP are not obtained by majority vote (as in NVP), or by detecting and discarding erroneous results (as in NSCP), but by $t/(n–1)$-diagnosis.

It could be argued that $t/(n–1)$-VP is only a variation of NVP; however, in our opinion, the majority voting check is an integral part of NVP, and each of N software versions in NVP is of equal importance. In marked contrast, the $t/(n–1)$-VP scheme does not try to find a majority of n results, but just to identify a presumably correct result. It can therefore deliver correct results with some probability even when the majority of results of n variants are incorrect. Moreover, $t/(n–1)$-VP has more flexible architectural features. In the architecture of Figure 1, the variant V_1 can be considered as being active, actually delivering the system output in the absence of faults; the variant V_4 and V_5 are used as "hot" spares, and V_2 and V_4 are only exploited for detecting errors and producing test outcomes. In addition, NVP requires that all variants should be designed to produce the results that are essentially identical. This constraint can be loosened in the $t/(n–1)$-VP approach. While the primary variant V_1 in the 1/(5–1)-VP architecture should attempt to produce the desired output, the spare variant V_5 may only attempt to provide a degraded service. In this form, the $t/(n–1)$-VP architecture can be used to implement a type of graceful degradation.

In principle, $t/(n–1)$-VP is also different from NVS (a form of sequential NVP) [10]. The $t/(n–1)$-VP method is based on so-called hot-standby redundancy, whereas NVS utilizes the cold-standby technique. More precisely, in the case that the results of the first two variants disagree (assuming $N = n = 3$), $t/(n–1)$-VP will select the result of the third variant, which has been available, as the system output through the result switch. NVS however has to first execute the third variant on the same set of input values and then makes a further decision by searching for a majority of the results. This validation process requires the extra execution time for the third variant and for the final decision. Clearly, in comparison with our scheme, NVS has relatively poor predictability of task completion time and may be inappropriate for certain time-critical applications.

It could be argued that $t/(n–1)$-VP is somewhat similar to NSCP. However, a fundamental distinction between the two schemes concerns their capacity for tolerating related faults. NSCP will fail (and even cause catastrophic consequences) whenever the two variants that form the active self-checking component produce the identical, but incorrect results (no matter how many spares are still available). In contrast, the $t/(n–1)$-VP scheme can tolerate up to t (independent or related) faults; that is, it can deliver the correct service even if t faulty variants compute identical incorrect results.

Finally, from the previous overview of software fault tolerance schemes, it is evident that the $t/(n–1)$-VP approach is quite distinct from the recovery block concept [2]. Like NVP and its variations, $t/(n–1)$-VP is complementary in many respects to RB. Recovery blocks can be more appropriate for those systems where hardware resources are limited and comparison-based adjudicators are inappropriate (a very detailed discussion of the relative advantages and disadvantages of NVP and RB is given in [1]). In the interests of simplicity and brevity, we will focus on the comparison of $t/(n–1)$-VP with NVP and NSCP without further discussing the recovery block approach.

3. DEPENDABILITY EVALUATION OF SOFTWARE FAULT TOLERANCE APPROACHES

In this section, we shall conduct a dependability evaluation of the $t/(n–1)$-VP scheme and the other similar approaches. Arlat, Kanoun and Laprie [16] recently analysed some special architectures using the RB, NVP and NSCP schemes respectively (mainly providing software redundancy able to tolerate single software faults). We exploit their modeling framework for investigating the software redundancy needed to tolerate two or more faults and establish a slightly different model to show the different impacts of independent and related faults on software dependability. Three architectures are analysed that can tolerate at least two software faults: $t/(n–1)$-VP and NVP using five variants, the former adopting a simple diagnosis algorithm for result selection (see table 1) and the latter employing the usual majority adjudication, and NSCP using six variants organized as three self-checking components. (Note that the NSCP architecture being considered here can tolerate two faults in most fault situations except the related faults that occur in an active self-checking component.) Expressions for F_X and C_X, where $X \in \{t/(n\text{-}1)\text{-VP, NVP, NSCP}\}$, will be derived using a Markov approach.

3.1. Underlying Assumptions

1) During the execution of the X scheme, related faults manifest themselves in the form of similar errors, whereas independent faults only cause distinct errors; and furthermore similar errors lead to common-mode failures, and distinct errors only cause independent failures;

2) all variants have the same probability of fault manifestation (or error);

3) only a single fault type, either independent or related, may appear during the execution of the scheme and no compensation [28] may occur between errors of the variants and of the adjudicator, *ie*, either an error is detected or it causes an incorrect output;

4) probabilities of independent and related faults are significantly low such that the probability p can be approximated to 1 (as assumed by others in similar settings; see [16] for example).

It is worth notice that these assumptions are used only to simplify the notation and the complexity in modeling and should not alter the significance of analytical conclusions to be derived. In particular, assumption 2) can be easily generalized to the case where the variants have respective fault characteristics. More complex models can be developed without applying assumption 4), *ie*, probabilities of independent and related faults are allowed to be arbitrary (descriptions about these models are provided in [17, 27]).

3.2. Detailed Reliability and Safety Models

We consider in this paper two different but complementary attributes of dependability: the continuity of service and the non-occurrence of catastrophic failure [28]. In general, we define software reliability as a measure of the time to failure and its safety as a measure of the time to catastrophic failure [16, 28]. The time (or the specified exposure period) in this definition is a relative concept and may mean a single run, a number of runs, or time expressed in calendar or execution time units of software. In the case of multiple runs, software may be idle between its executions. However software faults can manifest themselves only when software is executed. We will therefore focus on the execution process of software. Figure 2 shows a slight variation of the software behaviour model proposed by Arlat *et al.* [16]. In this behaviour model, a detected failure (*ie*, no service is delivered) is classified as *benign*; an undetected failure (*ie*, an incorrect result is delivered) can be either benign, or *catastrophic*. Since several runs are possible, service delivery may be restored from benign failures. Note that transitions from D and B to I and from U to B or C are applied only to the safety evaluation. Based on a Markov approach to modeling, reliability of the X approach can be evaluated simply by :

$$R_X(t) = e^{-(\sigma F_X)t}$$

where σ is the departure rate from state I, F_X the probability of failure of the X approach, and t the specified exposure time (for a detailed discussion of this formula see [16]);

and safety by:

$$S_X(t) = e^{-(\sigma C_X)t}$$

where σ is the departure rate from state I, C_X the probability of catastrophic failure of the X approach, and t the specified exposure time.

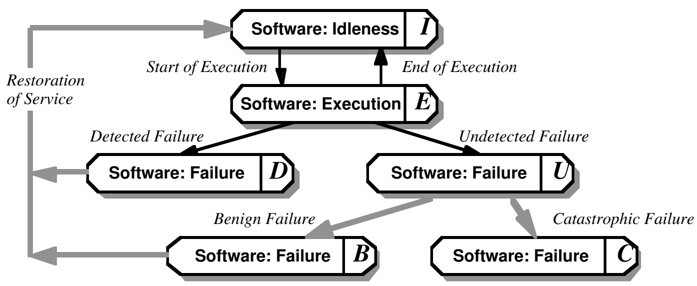

Figure 2. A modified behaviour model.

3.3. The $t/(n–1)$-VP Model for the 2/(5–1)-Architecture

Figure 3 describes a state-transition diagram for the 2/(5–1)-architecture based on the notation introduced at the end of the Introdoction. Respectively, execution states of the adjudicator are explained as follows.

1) State $A1$ corresponds to the case in which five variants produce the same correct results. According to assumption 4), probability p that all variants produce

correct results can be approximated to $1 - 5q_I - 10(q_I)^2 - 10(q_I)^3 - 5(q_I)^4 - (q_I)^5 - 10q_{2V} - 10q_{3V} - 5q_{4V} - q_{5V} - q_{AV1}$ (≈ 1). Given no fault in any variant, different types of adjudicator failure will lead to states D and U with respective probabilities q_{AD} and q_{AU}.

2) States $A2$ and $A3$ indicate activation of one or two independent faults in variants given no related fault among the variants. These fault types can be tolerated by this 2/(5–1)-architecture.

3) States $A4$, $A5$ and $A6$ correspond to cases in which three or more independent faults manifest themselves in variants. Since the number of faults has exceeded the bound 2, these states may lead to a failure state. However, through a more precise analysis, it is found that $t/(n$–1)-VP can still deliver a correct result in some situations (see a further discussion below).

4) State $A7$ represents activation of related faults in any two variants. These faults can be tolerated.

5) States $A8$, $A9$ and $A10$ correspond to cases in which related faults manifest themselves in more than two variants, which are undetectable.

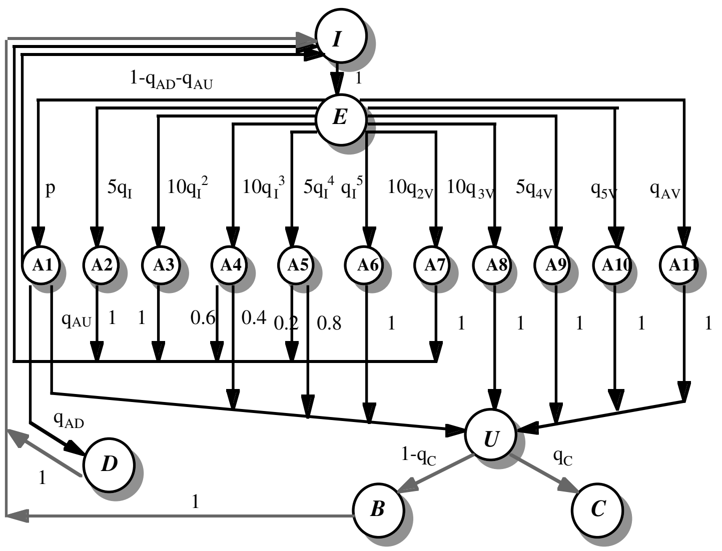

Figure 3. The $t/(n$–1)-VP model.

6) State $A11$ corresponds to activation of related faults between the adjudicator and the variants. This is also regarded as undetectable (see assumption 3).

In this 2/(5–1)-VP model, there is the transition from state $A4$ (or $A5$) to state I, that is, the architecture considered may still select a correct result as the system output even in the presence of more than two faults. Without loss of generality, take state $A4$ as an example. If three independent faults affect only three of variants V_1, V_2, V_3, and V_4, by assumption 1) their results will generate the syndrome where $\omega_{12} = \omega_{23} = \omega_{34} = 1$. The result of V_5 (a correct result) will then be chosen as the system output. Note that this class of events may occur with the probability $4q_I^3$. Similarly, if three independent faults affect only V_1, V_2 and V_5 (or only V_3, V_4 and V_5), according to table 1, the selected result can be still a correct one, with the probability $2q_I^3$. To sum up, the conditional probability of the transition from state $A4$ to state I is $(4q_I^3 + 2q_I^3) / (10q_I^3) = 0.6$. Therefore, the transition from A4 to a failure state can actually take place with the conditional probability $(4q_I^3) / (10q_I^3) = 0.4$.

From the state-transition diagram, it follows that

$$F_{t/(n-1)\text{-}VP} = p(q_{AD} + q_{AU}) + 4(q_I)^3 + 4(q_I)^4 + (q_I)^5 + 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV1}$$

A close but pessimistic approximation can be:

$$F_{t/(n-1)\text{-}VP} = q_{AD} + q_{AU} + 4(q_I)^3 + 4(q_I)^4 + (q_I)^5 + 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV1} \quad (1)$$

For evaluation of safety, only state C is absorbing; we thus have:

$$C_{t/(n-1)\text{-}VP} = q_C[q_{AU} + 4(q_I)^3 + 4(q_I)^4 + (q_I)^5 + 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV1}] \quad (2)$$

3.4. The NVP Model for the 5VP-Architecture

The NVP model for the 5VP-architecture is shown in Figure 4. The detailed analysis is essentially similar to that made for $t/(n–1)$-VP. A major difference is the case where multiple independent faults have an impact on three or more variants. In NVP, this case is much simpler — these faults will always lead to state D, assuming they are always detectable (but not tolerated). Thus, for reliability, F_{NVP} will be greater than $F_{t/(n-1)\text{-}VP}$:

$$F_{NVP} = q_{AD} + q_{AU} + 10(q_I)^3 + 5(q_I)^4 + (q_I)^5 + 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV2} \quad (3)$$

However, due to the detectability of multiple independent faults we have for safety:

$$C_{NVP} = q_C[q_{AU} + 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV2}] \quad (4)$$

which is obviously lower than $C_{t/(n-1)\text{-}VP}$.

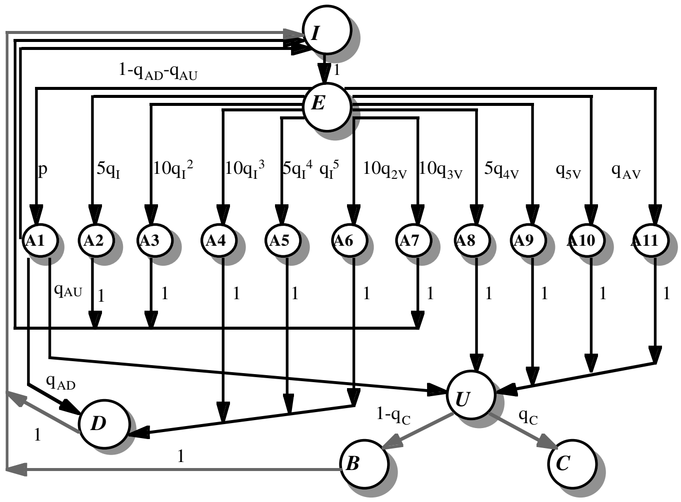

Figure 4. The NVP model.

3.5. The NSCP Model for the 3SCP-Architecture

Figure 5 shows the NSCP model for the 3SCP-architecture. The interpretations of the states are similar to those of the $t/(n-1)$-VP model though there are thirteen states $A1 \sim A13$ to consider because of the use of six variants. Independent faults in one or two of variants can be tolerated. Independent faults in three or more variants can be either tolerated or detected, as indicated by states $A4$ and $A5$. This shows that the NSCP scheme is quite effective on the treatment of independent faults.

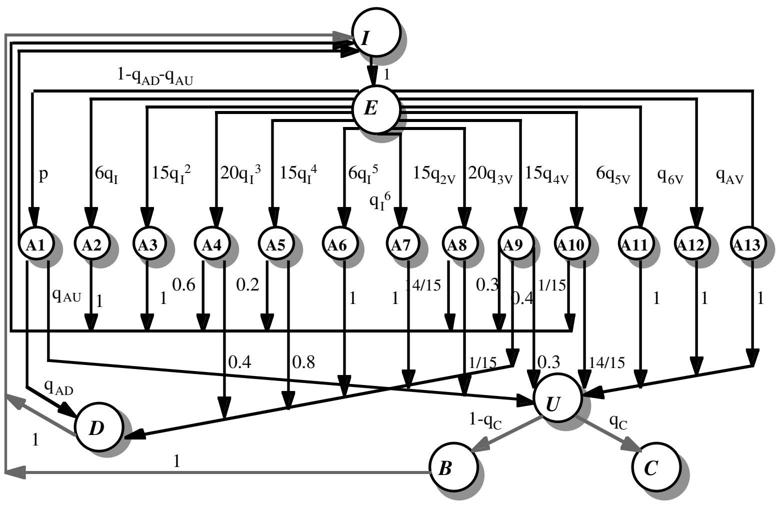

Figure 5. The NSCP model.

However, cases where related faults manifest themselves among multiple variants become more complicated. On one hand, NSCP is not fault-tolerant in the worst case — any related faults in active self-checking components could lead to certain failure states. On the other hand, some related faults can be tolerated or detected if they do not affect the pair of variants in an active self-checking component. Consider a representative case, state $A9$, in which related faults manifest themselves in three of the six software variants. There are three sub-cases to consider.

1) If related faults only occur in the spare self-checking components or such faults affect just a variant in the active component but not affect the first spare component, the 3SCP architecture can select a correct result and provide normal service. The conditional probability that this sub-case occurs is $[(6/20) \times (20q_{3V})] / (20q_{3V}) = 0.3$.

2) If related faults affect exactly one variant in every self-checking component, they can be detected effectively; the corresponding conditional probability is $[(2^3/20) \times (20q_{3V})] / (20q_{3V}) = 0.4$.

3) The worst sub-case is that related faults have an influence upon the pair of variants in the active component or an impact on the pair of variants in the first spare component given these related faults have affected a variant in the active one. In this sub-case, the 3SCP architecture will produce incorrect outputs, and the corresponding conditional probability is $[(6/20) \times (20q_{3V})] / (20q_{3V}) = 0.3$.

A similar analysis can be applied to other states. It therefore follows from the state-transition diagram:

$$F_{NSCP} = q_{AD} + q_{AU} + 8(q_I)^3 + 12(q_I)^4 + 6(q_I)^5 + (q_I)^6 + q_{2V} + 14q_{3V} + 14q_{4V} + 6q_{5V} + q_{6V} + q_{AV3} \quad (5)$$

Since independent faults can be either tolerated or detected, safety of the NSCP architecture concerns only related faults:

$$C_{NSCP} = q_C[q_{AU} + q_{2V} + 6q_{3V} + 14q_{4V} + 6q_{5V} + q_{6V} + q_{AV3}] \quad (6)$$

3.6. Remarks

From table 2 that summarizes the specific expressions for q_I's and q_U's, it is found that independent failures of the variants have a relatively small influence upon $t/(n-1)$-VP, but a larger impact on NVP and even more on NSCP. This is because the $t/(n-1)$-VP scheme possesses one of the significant characteristics of the $t/(n-1)$-diagnosis technique; namely, it is still possible in some fault situations for our proposed scheme to identify the correct results though faulty variants are in the majority.

Table 2. Specific expressions for q_I's and q_U's.

Parameters	$t/(n-1)$-VP	NVP	NSCP
$q_{I,X}$	$4(q_I)^3 + 4(q_I)^4 + (q_I)^5$	$10(q_I)^3 + 5(q_I)^4 + (q_I)^5$	$8(q_I)^3 + 12(q_I)^4 + 6(q_I)^5 + (q_I)^6$
$q_{A,X}$	q_{A1}(comparators)	q_{A2}(voter)	q_{A3}(comparator)
$q_{AV,X}$	q_{AV1}	q_{AV2}	q_{AV3}
$q_{mV,X}$	$10q_{3V} + 5q_{4V} + q_{5V}$	$10q_{3V} + 5q_{4V} + q_{5V}$	$q_{2V} + 6q_{3V} + 14q_{4V} + 6q_{5V} + q_{6V}$
$q_{U,X}$	$q_{AU} + 4(q_I)^3 + 4(q_I)^4 + (q_I)^5 + 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV1}$	$q_{AU} + 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV2}]$	$q_{AU} + q_{2V} + 6q_{3V} + 14q_{4V} + 6q_{5V} + q_{6V} + q_{AV3}$

As assumed previously, the adjudicators used in the three specific architectures are: result comparison plus a diagnosis algorithm in $t/(n-1)$-VP, a voter in NVP, and result comparison (plus the result switch) in NSCP. According to their complexities, it would be reasonable to rank q_A's and q_{AV}'s as follows.

$$q_{A(\text{NSCP})} \leq q_{A(t/(n-1)\text{-VP})} \leq q_{A(\text{NVP})} \tag{7}$$

$$q_{AV(\text{NSCP})} \leq q_{AV(t/(n-1)\text{-VP})} \leq q_{AV(\text{NVP})} \tag{8}$$

It follows from the table that related faults among variants have the same influence upon $t/(n-1)$-VP and NVP, but more serious on NSCP. This is a consequence of the fact that result comparison used in the self-checking components and the NSCP architecture itself are not effective enough to detect (or further tolerate) the related faults that may affect both variants in a self-checking component. Generally, this cannot be overcome by incorporating more variants into a given architecture. In contrast, both $t/(n-1)$-VP and NVP can tolerate certain number of related faults under the same bound and furthermore their fault tolerance capability can be enhanced, at least in principle, by involving more software variants.

Summarizing, the analysis above could thus suggest the following general conclusions.

For reliability:

$$F_{t/(n-1)\text{-}VP} < F_{NVP} < F_{NSCP} \qquad (9)$$

The inequality (9) means that the $t/(n-1)$-VP architecture has the lowest probability of failure — equivalently, the highest reliability. Due to high detectability of independent faults, NVP is however less sensitive to undetected faults than $t/(n-1)$-VP. The probability $q_{U(t/(n-1)\text{-VP})}$ for $t/(n-1)$-VP looks relatively high since this scheme may fail to detect some independent faults when the bound on the number of faulty variants is violated. This probability could be reduced by using more software variants. Note that the probability $q_{U(\text{NSCP})}$ is high as well, but again the incorporation of more variants would be of no effect on safety enhancement of NSCP. So for safety:

$$C_{NVP} < C_{t/(n-1)\text{-}VP} \leq C_{NSCP} \qquad (10)$$

Finally, it is important to notice that the evaluation data obtained here are used only to uncover the relative advantages and disadvantages of these schemes under consideration. For a given design using a particular scheme, the evaluation results also show how the design could be modified to further improve its dependability. Since the notion of software dependability captures many different concerns, including the qualities of reliability and safety, our analysis demonstrates the need of a delicate balance between these complementary attributes. In practice, a software designer must make an objective decision as to which technique is likely to be most appropriate for a specific application.

ACKNOWLEDGMENT

This research was supported by the ESPRIT Basic Research Actions 3092 and 6362 on Predictably Dependable Computing Systems (PDCS and PDCS2) and by the ESPRIT Long Term Research Project 20072 on Design for Validation (DeVa).

REFERENCES

[1] P. A. Lee, T. Anderson, *Fault Tolerance: Principles and Practice*, (2nd Edition) 1990; Prentice-Hall.

[2] B. Randell, “System structure for software fault tolerance”, *IEEE Trans. Software Engineering*, vol SE-1, 1975 Jun, pp 220-232.

[3] T. Anderson, “A structured decision mechanism for diverse software”, *Proc. Int.l Symp. Reliability in Distributed Software and Database Systems*, 1986, pp 125-129, Los Angeles, California.

[4] A. Avizienis, L. Chen, “On the implementation of *N*-version programming for software fault tolerance during program execution”, *Proc. Int'l Conf. Computer Software and Applications*, 1977, pp 149-155; New York.

[5] J.-C. Laprie, J. Arlat, C. Beounes, K. Kanoun, C. Hourtolle, “Hardware and software fault tolerance: Definition and analysis of architectural solutions”, *Proc. Int.l Symp. Fault-Tolerant Computing*, 1987 Jul, pp 116-121; Pittsburgh.

[6] R. K. Scott, J. W. Gault, D. F. McAllister, “Fault-tolerant software reliability modeling”, *IEEE Trans. Software Engineering*, vol SE-13, 1987 May, pp 582-592.

[7] K. H. Kim, H. O. Welch, “Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications”, *IEEE Trans. Computers*, vol 38, 1989 May, pp 626-636.

[8] B. Randell, J. Xu, “The evolution of the recovery block concept”, *Software Fault Tolerance* (M. Lyu Ed), (*Trends in Software* series) 1994, pp 1-22; John Wiley & Sons Ltd.

[9] B. W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, 1989; Addison-Wesley Publ. Co.

[10] A. Grnarov, J. Arlat, A. Avizienis, “On the performance of software fault-tolerance strategies”, *Proc. Int.l Symp. Fault-Tolerant Computing*, 1980 Oct, pp 251-253; Kyoto, Japan.

[11] J.-C. Laprie, J. Arlat, C. Beounes, K. Kanoun, “Definition and analysis of hardware and software fault-tolerant architectures”, *IEEE Computer*, vol 23, 1990 Jul, pp 39-51.

[12] S. E. Kreutzer, S. I. Hakimi, “System-level fault diagnosis: A survey”, *Microprocessing and Microprogramming*, vol 20, 1987 May, pp 323-333.

[13] M. Barborak, M. Malek, A. Dahbura, “The consensus problem in fault-tolerant computing”, *ACM Computing Surveys*, vol. 25, 1993 Jun, pp 171-220.

[14] J. C. Knight, N. G. Leveson, L. D. St. Jean, “A large scale experiment in *N*-version programming”, *Proc. Int.l Symp. Fault-Tolerant Computing*, 1985 Jul, pp 135-139; Ann Arbor.

[15] J. C. Knight, N. G. Leveson, “An experimental evaluation of the assumption of independence in multiversion programming”, *IEEE Trans. Software Engineering*, vol SE-12, 1986 Jan, pp 96-109.

[16] J. Arlat, K. Kanoun, J.-C. Laprie, “Dependability modeling and evaluation of software fault-tolerant systems”, *IEEE Trans. Computers*, vol 39, 1990 Apr, pp 504-513.

[17] A. T. Tai, J. F. Meyer, A. Avizienis, “Performability enhancement of fault-tolerant software”, *IEEE Trans. Reliability*, vol 42, 1993 Jun, pp 227-237.

[18] A. Friedman, “A new measure of digital system diagnosis”, *Proc. Int.l Symp. Fault-Tolerant Computing*, 1975 Jun, pp 167-170; Paris, France.

[19] S. Karunanithi, A. D. Friedman, “Analysis of digital systems using a new measure of system diagnosis”, *IEEE Trans. Computers*, vol 28, 1979 Feb, pp 121-133.

[20] J. Xu, S. Huang, “A new comparison-based scheme for multiprocessor fault tolerance”, *Microprocessing and Microprogramming*, vol 30, 1990 Aug, pp 617-623.

[21] J. K. Lee, J. T. Butler, "A characterization of *t*/*s*-diagnosability and sequential *t*-diagnosability in designs", *IEEE Trans. Computers*, vol 39, 1990 Oct. pp 1298-1304.

[22] J. Xu, "Fault tolerance based on system diagnosis techniques", *Proc. Open Workshop Predictably Dependable Computing Systems*, vol 1, 1991 May, pp 1-23; Newcastle, England.

[23] J. Xu, "The *t*/(*n*–1)-diagnosability and its applications to fault tolerance", *Proc. Int.l Symp. Fault-Tolerant Computing*, 1991 Jun, pp 496-503; Montreal, Canada.

[24] J. Xu, S. Huang, "Sequentially *t*-diagnosable systems: A characterization and Its applications", *IEEE Trans. Computers*, vol 44, (*Special Issue on Fault-Tolerant Computing*) 1995 Feb, pp 340-346.

[25] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostack and C. B. Weinstock, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control", *Proc. IEEE*, vol 66, 1978 Oct, pp 1255-1268.

[26] F. Cristian, "Exception handling and software fault tolerance", *IEEE Trans. Computers*, vol 31, 1982 Jun. pp 531-540.

[27] J. Xu, A. Bondavalli, F. Di Giondomenico, "Dynamic adjustment of dependability and efficiency in fault-tolerant software", *Predictably Dependable Computing Systems* (B. Randell *et al*., Ed), (*ESPRIT Basic Research* series) 1995, pp 155-172; Springer-Verlag.

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