Software Fault Tolerance: t/(n-1)-Variant Programming

J. Xu and B. Randell

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 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 existing software fault tolerance schemes. The results drawn from the comparison clearly show that t/(n-1)-VP is a viable addition or alternative to present techniques.

Classical researches on the dependability analysis of software fault tolerance approaches have almost been based on the simplest examples of architectures which can only tolerate single software faults, without considering tolerance to multiple or related faults. The conclusions from some such analysis are thus restricted. The dependability evaluation carried out in this paper deals with more complicated and general software redundancy, i.e., 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 n version programming approach (NVP) have the ability to tolerate some related faults between software variants; in general, t/(n-1)-VP would have the highest reliability, whereas NVP would be the best from the safety point of view.

© 1992 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details

**XU, Jie**

Software Fault Tolerance: $t/(n-1)$-Variant Programming
[By] J. Xu and B. Randell

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

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no.388)

**Added entries**

UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series 388
RANDELL, Brian

**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 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 existing software fault tolerance schemes. The results drawn from the comparison clearly show that $t/(n-1)$-VP is a viable addition or alternative to present techniques.

Classical researches on the dependability analysis of software fault tolerance approaches have almost been based on the simplest examples of architectures which can only tolerate single software faults, without considering tolerance to multiple or related faults. The conclusions from some such analysis are thus restricted. The dependability evaluation carried out in this paper deals with more complicated and general software redundancy, i.e., 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 $n$ version programming approach (NVP) have the ability to tolerate some related faults between software variants; in general, $t/(n-1)$-VP would have the highest reliability, whereas NVP would be the best from the safety point of view.

**About the author**

Mr. Jie Xu joined the Computing Laboratory in April 1990, where he is a Research Associate.

Professor Randell has been a Professor of Computing Science at the Computing Laboratory of the University of Newcastle upon Tyne since 1969.

**Suggested keywords**

RELATED SOFTWARE FAULTS
SOFTWARE FAULT TOLERANCE
SOFTWARE RELIABILITY
SOFTWARE SAFETY
SYSTEM-LEVEL FAULT DIAGNOSIS

**Suggested classmarks** (primary classmark underlined)
Dewey (18th): 001.6425 U.D.C. 681.322.06
Software Fault Tolerance: $t/(n-1)$-Variant Programming

Jie Xu and Brian Randell
Computing Laboratory
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, 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 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 existing software fault tolerance schemes. The results drawn from the comparison clearly show that $t/(n-1)$–VP is a viable addition or alternative to present techniques.

Classical researches on the dependability analysis of software fault tolerance approaches have almost been based on the simplest examples of architectures which can only tolerate single software faults, without considering tolerance to multiple or related faults. The conclusions from some such analysis are thus restricted. The dependability evaluation carried out in this paper deals with more complicated and general software redundancy, i.e., 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 $n$ version programming approach (NVP) have the ability to tolerate some related faults between software variants; in general, $t/(n-1)$–VP would have the highest reliability, whereas NVP would be the best from the safety point of view.

Key Words – Related software faults, Software fault tolerance, Software reliability, Software safety, System–level fault diagnosis.
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 [15]. Software fault tolerance is always based on the principle of design diversity. Design diversity is the approach in which two or more variants of a software module or a software system 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 it involves at least two variants of a system, tolerance to design faults necessitates an adjudicator [2] (or a decision algorithm) that determines a single (assumed to be) error-free result from one of the variants. Several techniques have been proposed for structuring a software system, and providing software fault tolerance: recovery blocks (RB)[16], $N$-version programming (NVP)[4], $N$ self-checking programming (NSCP)[13], the certification trail scheme (CT)[18] and some intermediate or combined techniques [9, 17]. These proposed techniques should be regarded as complementary to (not as a substitute for) verification & validation, software testing, and proof methodology, which aim at delivering fault-free software products.

The first scheme developed for achieving software fault tolerance was the recovery block scheme [1, 16], in which variants are organized in a manner similar to the standby sparing technique used in hardware. RB performs software design fault detection during runtime by augmenting any conventional hardware and software error detection mechanism with an acceptance test applied to the results of each variants. 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. In NVP, $N$-versions (variants) of a program which 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 (i.e. the minority) and pass on the (presumed to be correct) results (i.e. the majority). In simple cases the voting can be based on tests for identically; in general, a more sophisticated and application-oriented test is needed. Laprie et al. in 1987 identified the third type of scheme: $N$-self-checking programming. NSCP attains software fault tolerance by the parallel execution of $N$ self-checking software components. Each self-checking component is constructed from the association of
a pair of variants with a comparator. One component is regarded as the active component, and the others are considered as "hot" stand-by spares. Upon failure of the active component, service delivery is switched to a "hot" spare. Recently, Sullivan and Masson have developed an algorithm-oriented scheme, based on the case of what they term Certification Trails. The central idea of their method is to execute an algorithm so that it leaves behind a trail of data (certification trail) and, by using this data, to execute another algorithm for solving the same problem more quickly. Then, the outputs of the two executions are compared and considered correct only if they agree. If they disagree, either an exception will be signalled, or the third algorithm be executed.

It must be recognised that the success of a fault tolerance scheme depends to a great extent on its adjudicator and unreliability in the adjudicator can have a dramatic impact on the overall system reliability. Hence, adjudication mechanisms must themselves have high reliability. The design for a highly reliable adjudicator 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 that their reliability can be guaranteed and the overall performance of the system not be degraded significantly. In the RB scheme, 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 very difficult to ensure that the acceptance test and variants will be independent of each other. To overcome this problem, most of the other schemes are based on the notion of result comparison. However, the previously reported solutions are not entirely satisfactory. While the complexity of mechanisms used in NVP might impact the reliability and performance of the system, adjudicators constructed in NSCP and in CT are too simple to effectively detect and deal with related faults amongst variants, as will be discussed subsequently. We propose in the next section a new software tolerance scheme, called \( t/(n-1) \)-Variant Programming \( t/(n-1)\)-VP, which exploits a particular system diagnosis technique for constructing a simplified adjudication mechanism which is capable of tolerating multiple related faults.

Note also that it cannot be guaranteed that independently designed variants will fail independently (i.e., that faults in the different variants will occur at random and be unrelated) despite the adoption of the design diversity approaches [10, 11]. The reliability analysis of software-fault-tolerant techniques must thus include the effect of related (or
dependent) faults. A number of papers devoted to the dependable analysis of software fault tolerance approaches have appeared in the literature [3, 10, 17]. In particular, Arlat, Kanoun, and Laprie in 1990 presented a detailed evaluation of both reliability and safety measures. But, their analysis just concerned basic architectural examples tolerating single software faults so that their conclusions can hold only for those specific architectures. In the third section, we augment such published work by analysing more complex (more general) architectures that tolerate two or more software faults, identifying the ability of various approaches to tolerate independent and related faults, and distinguishing clearly related faults into the tolerated or detected fault class and the undetected fault class. The results drawn from our analysis provide designers with richer information about the fault tolerance properties of various architectures than the results from classical analysis, and point out that \( t/(n-1) \)-VP is a viable addition or alternative to present schemes for achieving software fault tolerance.

**Notation**

- \( C_X \) probability of catastrophic failure of the \( X \) approach
- \( F_X \) probability of failure of the \( X \) approach
- NSCP \( n \) self-checking programming
- NVP \( n \) version programming
- \( \text{Pr}(e|c) \) conditional probability that event \( e \) occurs under condition \( c \)
- \( R_X(t) \) reliability of the \( X \) approach
- \( S_X(t) \) safety of the \( X \) approach
- \( t/(n-1) \)-VP \( t/(n-1) \)-variant programming
- \( V_i \) \( i \)th software variant
- \( \omega \) (comparison) test outcome

Other, standard notation is given in "Information for Readers & Authors" at the rear of each issue.

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

In the theory of system-level fault diagnosis (see [6, 12] where further references can be found), a particular diagnosability measure, denoted as \( t/(n-1) \)-diagnosability, was first introduced in [7]. Its diagnosis goal is, for an \( n \)-unit computing system, 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 that is not in the set of size \( (n-1) \) exists so
that this unit can be unambiguously identified as fault-free, provided that the system 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 set of $n$ results generated by $n$ replicated software modules (of independent design). Our intuition indicates that we can benefit from the utilization of $t/(n-1)$-diagnosis since this diagnosis measure cuts down significantly the requirement on the number of tests (e.g., the number of result-comparison tests) relative to previous diagnosis schemes. It is thus possible to construct a simple, but dependable adjudication mechanism. Based on current theoretical results of $t/(n-1)$-diagnosis in [8, 14, 19, 20], we develop a new scheme for tolerating hardware and/or software faults. Our description of this scheme is first in terms of applications to software fault tolerance, but the approach can also be implemented with hardware [19]. Here, we only take into account software faults. Two classes of software faults are distinguished: independent faults and related faults. Independent faults occur in single variants or in the adjudication mechanism, whereas related faults can take place amongst several variants and amongst the adjudicator and one or more variants.

In what follows, we shall term such a software fault tolerance scheme $t/(n-1)$-Variant Programming ($t/(n-1)$-VP). A $t/(n-1)$-VP architecture can always identify at least one correct result from the subset of results of $n$ software modules (or variants), provided that the number of faulty modules in the architecture does not exceed $t$ (i.e., 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 modules (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 (e.g., through switching of the results); and if no acceptable result is identified, the system will invoke spare software variants, if exists any, or simply signal an exception.

Figure 1 shows a $t/(n-1)$-VP architecture where $n = 5$ and $t = 2^{1)}$. The $t/(n-1)$-VP

---

1) We use such a somewhat complicated example in order to show the ability of the proposed method to tolerate related faults. A basic architecture with $n = 3$ and $t = 1$ can be derived, requiring only one comparator.
Figure 1. A $t/(n-1)$-VP architecture with $n = 5$ and $t = 2$. 
architecture is thus composed 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 2 simultaneous faults. Three comparators $C_1$, $C_2$ and $C_3$ are placed at the outputs of the 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 $\omega_{12}$, $\omega_{23}$ and $\omega_{34}$ 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 $V_5$ depending on the value of the syndrome and then switches service delivery (i.e., the system output) to the selected result. The adjudicator of the architecture is implemented by the three comparators, the $t/(n-1)$-diagnostor and the output switch. Although 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, according to the conclusions in Theorem 4 and the table of Figure 8 in [19], we conclude that the architecture is $t/(n-1)$-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 2. In other words, besides tolerance to two independent faults in any two variants, the architecture can tolerate related faults between any two variants as well. The reason why it can tolerate at least two faults is further explained as follows.

Let $r_1$, $r_2$, ..., and $r_5$ be the results of variants $V_1$, $V_2$, ..., and $V_5$ respectively. The table of Figure 2 shows all possible syndromes and the corresponding results of variants which can be unambiguously diagnosed as correct (under the assumption that no more than two faults occur simultaneously). For example, in the case that $\omega_{12} = 0$, $\omega_{23} = 1$ and $\omega_{34} = 0$, we cannot identify a single correct result from amongst those produced by variants $V_1$, $V_2$, $V_3$ and $V_4$. However we can infer from the syndrome that two or more of the variants $V_1$, $V_2$, $V_3$ and $V_4$ have generated incorrect output results because one single fault cannot lead to such a syndrome. Hence, the result of $V_5$ must be "correct". When $\omega_{12} = \omega_{23} = \omega_{34} = 0$, 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. Based on the previous assumption that $t = 2$, it is therefore appropriate to identify them as correct. From this table, we find an interesting fact that for any syndrome, at least one of results $r_1$, $r_4$ and $r_5$ must be correct. Accordingly, the architecture can produce the system output by choosing just amongst the outputs of the three variants $V_1$, $V_4$ and $V_5$. In fact, the table can also be viewed as a simple diagnosis algorithm for the specific architecture.
<table>
<thead>
<tr>
<th>$\omega_{12}$</th>
<th>$\omega_{23}$</th>
<th>$\omega_{34}$</th>
<th>Presumably Correct Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$r_1$ $r_2$ $r_3$ $r_4$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$r_1$ $r_2$ $r_3$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$r_1$ $r_2$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$r_2$ $r_3$ $r_4$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$r_3$ $r_4$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$r_5$</td>
</tr>
</tbody>
</table>

Figure 2. Possible syndromes and result selections for the $r(=2)/(n-1=4)$–VP architecture.
Once that the number of the faulty variants in the architecture exceeds the previously determined bound $t$, the correctness of output results by the system will not be guaranteed, and what is worse, this specific fault situation cannot be detected completely by any fault tolerance scheme. To overcome this problem partly, exception handling techniques [5] can be used. In the $2/(4)$–architecture of Figure 1, for instance, exception-handlers can be added to variants each. The function of the exception handling in a single variant is to handle any errors which might be detected during the execution of the variant by signalling an exception to the diagnostor. The diagnostor comes to a final decision according to the value of the syndrome and all the exception signals received so far: either generating a presumably correct result or signalling an exception without delivering any result.

The $t/(n-1)$–VP scheme has resemblances with other fault tolerance techniques that have been previously proposed and examined, 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 timeout 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 still deliver correct results with some probability when the majority of results obtained by the system are incorrect. Moreover, $t/(n-1)$–VP can have more flexible redundancy features. In the architecture of Figure 3, the variant $V_1$ can be considered as being active, actually delivering output results in the absence of faults; the variants $V_4$ and $V_5$ are used as "hot" spares, and $V_2$ and $V_3$ are only exploited for detecting errors and producing test outcomes. In addition, NVP requires that all variants should be designed to produce the results which are essentially identical, whereas this constraint can be loosened in the $t/(n-1)$–VP approach. While the primary variant $V_1$ in the $1/(4)$–VP architecture should attempt to produce the desired outcome, 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 form of graceful degradation.
It could also 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 two variants in 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 where $t \leq (n-1)/2$. That is, it can deliver the correct service even if $t$ faulty variants compute identical incorrect results.

The aim of our work is to show how the $t/(n-1)$–diagnosis technique can be applied to the design of software fault tolerance approaches. Here, we will not further discuss details of the particular diagnosis technique. We presented in [19, 20] a set of theoretical results of $t/(n-1)$–diagnosability and several $t/(n-1)$–diagnosable configurations. Just regarding "units" in the Model for diagnosis as software modules (or variants), one can use immediately all results in [19, 20] for structuring software–fault–tolerant systems. For practical values of $n$ (the number of variants), a $t/(n-1)$–VP architecture requires only $O(n)$ comparators and 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 $O(n^2)$ time comparisons).

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 [3] recently analysed some special architectures (mainly providing software redundancy able to tolerate single software faults), using the RB, NVP and NSCP approaches. For our purposes, we shall exploit their modelling framework for considering the software redundancy needed to tolerate two or more faults and will establish a slightly different model to show the different impacts of independent or related faults on the dependability of $t/(n-1)$–VP, NVP, and NSCP. Let $F_X$ (respectively, $C_X$) be the probability of failure (respectively, catastrophic failure) of the X approach. Three architectures are analysed that can tolerate at least two software faults: $t/(n-1)$–VP (where $n = 5$ and $t = 2$), NVP (5 versions), and NSCP (3 self-checking components made up of 6 variants). Expressions for each $F_X$ and $C_X$ can be obtained using a Markov approach. To compute these expressions, the following notation is defined, where $Pr\{e | c\}$ represents the conditional probability that event $e$ occurs under condition $c$, and $X$ belongs to \{$t/(n-1)$–VP, NVP, NSCP\}.
Notation

\[ q_I = \Pr\{\text{activation of an independent fault in one variant of } X|\text{execution}\}, \]
\[ q_A = \Pr\{\text{activation of an independent fault in the adjudicator of } X|\text{execution}\} \]
\[ = q_{AD} + q_{AU} \text{ where} \]
\[ q_{AD} = \Pr\{\text{activation of a detected independent fault in the adjudicator of } X \]
\[ |\text{execution}\} \] and
\[ q_{AU} = \Pr\{\text{activation of an undetected independent fault in the adjudicator of } X \]
\[ |\text{execution}\} , \]
\[ q_{VA} = \Pr\{\text{activation of related faults between the variants and the adjudicator of } X|\text{execution}\} , \]
\[ q_{mV} = \Pr\{\text{activation of related faults between } m \text{ variants of } X|\text{execution}\} , \]
\[ q_{UD} = \Pr\{\text{undetected failure } | \text{ execution}\} , \]
\[ q_C = \Pr\{\text{undetected error causing a catastrophic failure}|\text{execution}\}. \]

Basic Assumptions

(i) Related faults manifest themselves under 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; (ii) the probability of fault activation is the same for all variants; (iii) just a single (either independent or related) fault type may manifest itself during the whole software execution (including variants and adjudicator) and no compensation may occur between the errors of the variants and of the adjudicator.

These assumptions are used only to simplify the notation and the complexity in computation, and do not alter the significance of the conclusions to be obtained; in particular, the conclusions can be directly generalized to the case of the variants which have respective fault characteristics.

Detailed Reliability and Safety Models

We will now consider two different but complementary attributes of dependability: the continuity of service and the non-occurrence of catastrophic failure. In general, we define reliability as a measure of the time to failure and safety as a measure of the time to catastrophic failure. Safety mainly concerns the utilization of computing systems in critical applications. A safe system might have to stop delivering output results in order to
Figure 3. A modified behaviour model.
avoid the occurrence of catastrophic failures. So, a reliable system is not inevitably safe and vice versa. Note that software faults can manifest themselves only when software is executed. Thus, a behaviour model can be created by means of slightly modifying the model proposed by Arlat et al. [3](see Figure 3). A detected failure of a system or an architecture can be regarded as benign because no acceptable result is identified by the adjudicator and no output result is delivered. An independent failure of the adjudicator that causes no output result is also termed as a detected failure. On the other hand, an undetected failure cannot be simply equivalent to a catastrophic failure: for example, an erroneous result would be delivered but no catastrophic consequences caused. We will thus divide undetected failures into two classes: benign failures and catastrophic failures. It is further assumed that service delivery can be restored from detected failures or benign undetected failures. Transitions from D or B to I and from U to B or C hold only for safety. From Figure 3, suppose that the departure rate from state I is \( \sigma \). Then, the reliability of the \( X \) approach can be evaluated by:

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

and the safety by:

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

In the following, we will demonstrate the production of reliability and safety expressions for \( t/(n-1) \)-VP, NVP, and NSCP.

**The \( t/(n-1) \)-VP Model**

Figure 4 gives the \( t/(n-1) \)-VP model. State E is the execution state of software. States \( A_1 \sim A_{11} \) correspond to the execution of the adjudicator (or the decision algorithm). Respectively,

1. state \( A_1 \) indicates the case of five variants which produce correct results:
   \[
p = 1 - 5q_I - 10(q_I)^2 - 10(q_I)^3 - 5(q_I)^4 - (q_I)^5 - 10q_{2V} - 10q_{3V} - 5q_{4V} - q_{5V} - q_{AV1},
   \]
   and \( q_{AD} \) and \( q_{AU} \) are the failure probability of the adjudicator, which lead to states D and U respectively;

2. state \( A_2 \) indicates activation of an independent fault in one of five variants, and state \( A_3 \) shows activation of two independent faults in two of five variants: both these fault types can be tolerated by the \( 2/(4) \)-architecture;
Figure 4. The $t/(n-1)$-VP model.
(3) states A₄, A₅ and A₆ indicate activation of multiple independent faults in three or more variants: some of them lead to state U, but the others can be tolerated despite the number of faults being greater than the bound 2;

(4) state A₇ represents activation of related faults in two of five variants: these related faults can be effectively tolerated;

(5) states A₈, A₉ and A₁₀ correspond to activation of related faults in three or more variants, which are undetectable; and

(6) state A₁₁ shows activation of related faults between the adjudicator and the variants, which is also regarded as undetectable.

From the model, note that state classes D and U are absorbing for reliability. We have:

\[ F_{t/(n-1)\text{-VP}} = p(q_{AD} + q_{AU}) + 4(q_i)^3 + 4(q_i)^4 + (q_i)^5 + 10q_{3V'} + 5q_{4V'} + q_{5V'} + q_{AV1} \]

A close but pessimistic approximation can be:

\[ F_{t/(n-1)\text{-VP}} = q_{AD} + q_{AU} + 4(q_i)^3 + 4(q_i)^4 + (q_i)^5 + 10q_{3V'} + 5q_{4V'} + q_{5V'} + q_{AV1} \]

For safety, only state class C is absorbing, and we thus obtain:

\[ C_{t/(n-1)\text{-VP}} = q_{C}[q_{AU} + 4(q_i)^3 + 4(q_i)^4 + (q_i)^5 + 10q_{3V'} + 5q_{4V'} + q_{5V'} + q_{AV1}] \]

The NVP Model

The NVP model is shown in Figure 5. A detailed analysis similar to that for the \( t/(n-1)\text{-VP} \) model can be conducted. Major differences include that multiple independent faults in three or more variants will always lead to state D (the detected failure state) and not be tolerated, but as these faults are detectable, the probability of catastrophic failure can be therefore decreased. For reliability, \( F_{NVP} \) will be greater than \( F_{t/(n-1)\text{-VP}} \), that is:

\[ F_{NVP} = q_{AD} + q_{AU} + 10(q_i)^3 + 5(q_i)^4 + (q_i)^5 + 10q_{3V'} + 5q_{4V'} + q_{5V'} + q_{AV2} \]

However, for safety, we have:

\[ C_{NVP} = q_{C}[q_{AU} + 10q_{3V'} + 5q_{4V'} + q_{5V'} + q_{AV2}] \]

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

The NSCP Model

Figure 6 shows the NSCP model. Due to the use of up to six software variants, the execution of the adjudication function corresponds to thirteen state classes A₁ ~ A₁₃. Independent faults in one or two of variants can be effectively tolerated by the 3SCP architec-
Figure 5. The NVP model.
Figure 6. The NSCP model.
ture and, moreover, some of multiple independent faults be tolerated as well, as indicated by state classes A₄ and A₅. However, the cases concerning related faults are more complicated. On one hand, NSCP is not (related) fault-tolerant in the worst case: any related faults amongst variants may lead to some failure states. On the other hand, some of the related faults can be tolerated or detected. Consider, for example, the state class A₀, which represents activation of related faults in three of the variants. There will be 30% of the faults that can be tolerated, 40% of them detected, and 30% of them undetected. From the model, it follows:

\[ F_{\text{NSCP}} = q_{\text{AD}} + q_{\text{AU}} + 8(q_{I})^3 + 12(q_{I})^4 + 6(q_{I})^5 + (q_{I})^6 \]
\[ + q_{\text{Y}} + 14q_{\text{Y}} + 14q_{\text{Y}} + 6q_{\text{Y}} + q_{\text{Y}} + q_{\text{AV}} \]

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

\[ C_{\text{NSCP}} = q_{C}[q_{\text{AU}} + q_{\text{Y}} + q_{\text{Y}} + 14q_{\text{Y}} + 6q_{\text{Y}} + q_{\text{Y}} + q_{\text{AV}}] \]

**Remarks**

The table of Figure 7 summarizes the specific expressions for \( q_I \)'s and \( q_{UD} \)'s, and it shows that independent failures of the variants have 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 the specific feature of system diagnosis techniques, namely, it is still possible to identify the correct results when the number of faulty variants exceeds the bound \( t \) occasionally.

Without loss of the generality, assume that adjudicators used in three approaches are respectively: comparators plus a diagnosis algorithm in \( t/(n-1) \)-VP, a voter in NVP and comparators (plus the result switch) in NSCP. Then the adjudication mechanism of NSCP would be the simplest. Therefore, it could be reasonable to rank \( q_A \)'s and \( q_{AV} \)'s as follows:

\[ q_A(\text{NSCP}) \leq q_A(\text{t/(n-1)-VP}) \leq q_A(\text{NVP}) \]
\[ q_{AV}(\text{NSCP}) \leq q_{AV}(\text{t/(n-1)-VP}) \leq q_{AV}(\text{NVP}) \]

Related faults among variants have the same influence upon \( t/(n-1) \)-VP and NVP, but more on NSCP. This is a consequence of the fact that the adjudication function performed by the NSCP adjudicator and the NSCP architecture itself are too simple to effectively tolerate or detect related faults. This cannot be overcome by using more variants. Both \( t/(n-1) \)-VP and NVP can tolerate related faults under the same bound on the number
<table>
<thead>
<tr>
<th>Parameters</th>
<th>$t/(n-1)-VP$</th>
<th>NVP</th>
<th>NSCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_{L,X}$</td>
<td>$4(q_l)^3 + 4(q_l)^4 + (q_l)^5$</td>
<td>$10(q_l)^3 + 5(q_l)^4 + (q_l)^5$</td>
<td>$8(q_l)^3 + 12(q_l)^4 + 6(q_l)^5 + (q_l)^6$</td>
</tr>
<tr>
<td>$q_{A,X}$</td>
<td>$q_{A1}$(comparators)</td>
<td>$q_{A2}$(voter)</td>
<td>$q_{A3}$(comparator)</td>
</tr>
<tr>
<td>$q_{AV,X}$</td>
<td>$q_{AV1}$</td>
<td>$q_{AV2}$</td>
<td>$q_{AV3}$</td>
</tr>
<tr>
<td>$q_{MV,X}$</td>
<td>$10q_{3V} + 5q_{4V} + q_{5V}$</td>
<td>$10q_{3V} + 5q_{4V} + q_{5V}$</td>
<td>$q_{2V} + 6q_{3V} + 14q_{4V} + 6q_{5V} + q_{6V}$</td>
</tr>
<tr>
<td>$q_{UD,X}$</td>
<td>$q_{AU} + 4(q_l)^3 + 4(q_l)^4 + (q_l)^5$</td>
<td>$q_{AU} + 10q_{3V} + 5q_{4V} + q_{AV1}$</td>
<td>$q_{AU} + q_{2V} + 6q_{3V} + 14q_{4V}$</td>
</tr>
<tr>
<td></td>
<td>$+ 10q_{3V} + 5q_{4V} + q_{5V} + q_{AV2}$</td>
<td>$+ q_{5V} + q_{AV2}$</td>
<td>$+ 6q_{5V} + q_{6V} + q_{AV3}$</td>
</tr>
</tbody>
</table>

Figure 7. Specific expressions for $q_l$'s and $q_{UD}$'s.
of faults and, furthermore, their ability to tolerate related faults can be enhanced by increasing the number of variants if required.

Summarizing, the analysis conducted in this section could suggest the following general conclusions.

For reliability:

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

This inequality means that the $i/(n-1)$-VP architecture has the lowest probability of failure; equivalently, the highest reliability. However, from the safety point of view, NVP is less sensitive to undetected faults than $i/(n-1)$-VP and NSCP. NVP has the most powerful capability of detecting independent faults. For $i/(n-1)$-VP, the probability $q_{UD(i/(n-1)-VP)}$ is relatively high; this is due to the fact that the $i/(n-1)$-VP architecture may fail to detect some independent failures when more than $i$ faults occur. There are two methods for lowering this probability: (i) to use more software variants; and (ii) to add Exception-handlers to variants each. Note that the probability $q_{UD(NSCP)}$ is also high, but the increase of the number of variants would be of no effect on improvement of the safety of NSCP. Therefore, for safety:

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

ACKNOWLEDGMENT

The authors would like to acknowledge the financial support of the ESPRIT Basic Research Action on Predictably Dependable Computing Systems.

REFERENCES


