

Contents

3 Architectural Issues in Software Fault Tolerance	47
3.1 INTRODUCTION	47
3.2 APPROACHES TO SOFTWARE-FAULT TOLERANCE	49
3.3 ANALYSIS OF SOFTWARE FAULT TOLERANCE	51
3.4 DEFINITION AND ANALYSIS OF HARDWARE-AND-SOFTWARE FAULT TOLERANT ARCHITECTURES	62
3.5 CONCLUSIONS	75



3

Architectural Issues in Software Fault Tolerance

JEAN-CLAUDE LAPRIE, JEAN ARLAT, CHRISTIAN BÉOUNES
and KARAMA KANOUN

LAAS-CNRS, France

ABSTRACT

This chapter presents a unified overview of architectural solutions to software fault tolerance by means of design diversity, i.e., the production of two or more systems aimed at delivering the same service through separate designs and realizations. The first part synthesizes the major approaches for software-fault tolerance; in addition to the recovery blocks (discussed in Chapter 1) and N -version programming (discussed in Chapter 2) methods, a third type of approach is identified from the careful examination of current, real-life systems: N self-checking programming. Then, the three approaches to software fault tolerance are analyzed with respect to two viewpoints: dependability and cost. Finally, architectures aimed at tolerating both hardware and software faults are defined and analyzed.

3.1 INTRODUCTION

There exist two largely differing approaches to software fault tolerance, depending on the forecast aim, either preventing a task failure to lead to the system complete disruption, or ensuring service continuity. In the former case, an erroneous task will be detected as soon as possible, and will be halted to prevent error propagation; this approach is often termed as *fail-fast* [Gra86, Gra90]. The latter case requires the existence of at least another component able to fulfill the task, designed and implemented separately from the same specification; this approach thus requires the use of *design diversity*. This chapter concentrates on software fault tolerance based on design diversity.

Design-fault tolerance by means of design diversity is a concept that traces back to the very early age of informatics: see the quotations relative to Babbage's work in [Avi79, Ran87].

Closer to us, after its (re-)inception in the 70's [Elm72, Fis75, Ran75, Che78], software fault tolerance by design diversity has become a reality, as witnessed by the real-life systems and the experiments reported in [Vog88]. The currently privileged domain where design diversity is applied is the domain of safety-related systems. In such systems, an extreme attention is paid to *design faults*, where the term "design" is to be considered in a broad sense, from the system requirements to realization, during initial system production as well as in possible future modifications. Design faults are indeed a source of *common-mode* failures, which defeat fault tolerance strategies based on strict replication (thus intended to cope with physical faults), and have generally catastrophic consequences.

Pre-computer realizations of safety-related systems were classically calling for *diversified designs*, i.e., the production of two or more *variants* of a system or equipment intended to fulfill the same function through implementations based on different technologies. The aim of diversified designs is to minimize as much as possible the likelihood of occurrence of common-mode failures; a typical example is provided by a hard-wired electronic channel backed by an electro-mechanic or electro-pneumatic channel. In addition, the systems architecture was based on the *federation* of equipment implementing each one or several subfunctions of the system; one of the criteria for partitioning the system global function into subfunctions was that a failure of any equipment should be confined, and should not prevent the global function of the system to be performed, possibly in a degraded mode. When the transition was made to computer technology, the safety-related systems generally kept on the federation approach [Avi86]. Each subfunction was then implemented by a "complete" computer: hardware, executive and application software. Examples of this approach may be found in airplane flight control systems (e.g., the Boeing 757/767 [Spr80]) or nuclear plant monitoring (e.g., the SPIN system [Rem82]). A pre-requisite to confining computer failures is the auto-detection of the error(s) having led to failure. The auto-detection of errors due to design faults can be achieved through two main approaches:

- acceptance tests of the results under the form of executable assertions [And79, Mah84], which constitute a generalization and a formalization of the likelihood checks which are classical in process control;
- diversified design leading to two software variants whose results are compared; real-life examples are the Airbus A300 and A310 [Mar82, Rou86, Tra88], or the Swedish railways' interlocking system [Ste78, Hag86].

The federation approach generally leads to a very large number of processing elements, larger than what would be necessary in terms of computing power; for instance, the Boeing 757/767 flight management control system is composed of 80 distinct functional microprocessors (300 when redundancy is accounted for). Although such large numbers of processing elements are affordable due to the low cost of today's hardware, this approach suffers severe limitations, because of the subfunctions isolation, which inhibits the cooperative use of total computer resources:

- limitations to evolving towards new functionalities, due to a restricted exploitation of the software possibilities;
- limitations to the improvement of reliability and safety, due to rigid redundancy management schemes, induced by the necessity of local redundancies for the equipment.

An additional step towards a better use of the possibilities offered by computers consists

in having several subfunctions implemented by software, supported by the same hardware equipment. Such an approach, which can be termed as integration, comes up against software failures, which are due to design faults only; adopting the integration approach thus requires software-fault tolerance. Moreover, a joint evolution of some safety-related systems (e.g., the flight control systems) is towards limited, if any, back-up possibilities, either manual or by non-computer technologies. This is of course an additional incitement to software fault tolerance, since a safe behavior of the system is then entirely depending on a reliable behavior of the software. Real-life examples are the NASA's space shuttle [She78, Cau81, Gar81], the Boeing 737/300 [You84], and the Airbus A320/A330/A340 [Tra88, Bri93].

This chapter elaborates on the work reported in [Lap87, Lap90, Lap94]. The remaining part of the chapter is composed of four Sections. Section 3.2 is devoted to a unified presentation of the approaches for software-fault tolerance; in addition to the recovery blocks and N -version programming methods, a third type of approach is identified from the careful examination of current, real-life systems: N self-checking programming. In Section 3.3, the three approaches to software fault tolerance are analyzed with respect to two viewpoints: dependability and cost. Section 3.4 is devoted to the definition and analysis of a set of hardware-and-software fault-tolerant architectures. Finally, Section 3.5 provides some concluding remarks.

3.2 APPROACHES TO SOFTWARE-FAULT TOLERANCE

Design diversity may be defined [Avi86] as the *production of two or more systems aimed at delivering the same service through separate designs and realizations*. The systems produced through the design diversity approach from a common service specification are termed *variants*. Besides the existence of at least two variants of a system, tolerance of design faults necessitates a *decider*, aimed at providing an — assumed as being — error-free result from the variant execution; the variant execution has to be performed from consistent initial conditions and inputs. The common specification has to address explicitly the *decision points*, i.e., a) when decisions have to be performed, and b) the data upon which decisions have to be performed, thus the data processed by the decider.

The two most well documented techniques for tolerating software design faults are the *recovery blocks* and the *N -version programming* that are described in detail in the two preceding chapters; we simply recall the main features of these techniques.

In the recovery block (RB) approach [Ran75, And81], the variants are termed alternates and the decider is an acceptance test, which is applied sequentially to the results provided by the alternates: if the results provided by the primary alternate do not satisfy the acceptance test, the secondary alternate is executed, and so on. In the N -version programming (NVP) approach [Che78, Avi85], the variants are termed versions, and the decider performs a vote on all versions' results.

The term *variant* is preferred here as a unifying term to the terms *alternate* or *version* for the following reasons:

- the term "alternate" reflects sequential execution, which is a feature specific to the recovery block approach;
- the term "version" has a widely accepted different meaning: successive versions of a system resulting from fault removal and/or functionality evolution; during the life of a diversity designed system, several versions of the variants are expected to be generated.

The RB and NVP approaches have led to a number of variations, such as the consensus recovery block [Sco87] that combines the decision procedures of NVP and RB, the $t/(n-1)$ -variant programming scheme [Xu91] that uses a system-level diagnosis technique inherited from hardware diagnosis theory to isolate the faulty variant(s), or the self configuring optimistic programming technique [Bon93] that provides a trade-off between space and time redundancy needed to implement and execute the software variants.

The hardware-fault tolerant architectures equivalent to RB and to NVP are *stand-by sparing* (also termed as passive dynamic redundancy) and *N-modular redundancy* (also termed as static redundancy), respectively. The equivalent of a third approach to hardware-fault tolerance, *active dynamic redundancy* (very popular especially when based on self-checking components, as in the AT&T ESSs, Stratus systems, etc.), does not appear to have been described in the published literature as a generic technique for software-fault tolerance. However, self-checking programming has long been introduced (see, e.g., [Yau75]) where a self-checking program results from the addition of redundancy into a program to check its own dynamic behavior during execution. A self-checking software component is considered as resulting either from the association of an acceptance test to a variant, or from the association of two variants with a comparison algorithm. Fault tolerance is provided by the parallel execution of $N \geq 2$ self-checking components. At each execution of the system so constituted, a self-checking component is considered as being acting (in the sense of the delivered service, and thus of the effective delivery of the results to the controlled or monitored application); the other self-checking components are considered as — "hot" — spares. Upon failure of the acting component, service delivery is switched to a self-checking component previously considered as a spare; upon failure of a spare component, service keeps being delivered by the acting component. Error processing is thus performed through error detection and — possible — switching of the results. In what follows, we shall term such a software fault tolerance approach *N self-checking programming* (NSCP).

Our aim is not so much introducing a new software fault tolerance approach as providing a clear classification of the various approaches that may be considered. In fact, most of the real-life systems mentioned in the introduction do not actually implement either RB or NVP, but are based on self-checking software. For instance:

- the computer-controlled part of the flight control systems of the Airbus A300 and A310 and the Swedish railways' interlocking system are based on parallel execution of two variants whose results are compared, and they stop operation upon error detection;
- the flight control system of the Airbus A320 is based on two self-checking components, each of them being in turn based on parallel execution of two variants whose results are compared: tolerance to a single fault needs four variants¹.

Four additional comments on NSCP:

- when a self-checking software component is based on the association of two variants, one of the variants only may be written for the purpose of fulfilling the functions expected from the component, the other variant being an extended acceptance test; examples of such an approach are a) performing computations with a different accuracy, b) inverse function

¹ It is worth mentioning that the two self-checking components (each with two variants) do not exactly deliver the same service: when switching from the initially acting component to the other one, the critical functions are preserved whereas the non-critical functions are performed in a degraded mode.

- when possible, or c) exploiting some intermediate results of the main variant as in the "certification-trail" approach [Sul90];
- as in N -version programming, the fact that the components are being executed in parallel² necessitates an input consistency mechanism;
- the acceptance tests associated to each variant or the comparison algorithms associated to a pair of variants can be the same, or specifically derived for each of them — from a common specification;
- it could be argued that NSCP is "parallel recovery block"; in our opinion, the RB concept cannot be reduced to the association of alternates together with an acceptance test: the — backward — recovery strategy is an integral part of it.

Figure 3.1 gives Petri-net execution models for the three identified strategies for software-fault tolerance. This figure is relative to the layer where diversification is performed, and thus deliberately omits some mechanisms, e.g., a) recovery points establishment in RB, b) input data consistency in NSCP and NVP, or c) replication of the decision algorithm (vote) on each of the sites supporting the versions in NVP. The most significant features of the strategies clearly appear on this figure:

- the differences in the execution schemes, i.e., sequential for RB and parallel for NSCP and NVP;
- the responsibility of each self-checking component in NSCP for the delivery or not of an acceptable result, whereas the judgment on result acceptability is cooperative in NVP.

Figure 3.2 summarizes the main characteristics of the strategies. Of particular interest, concerning the selection of a strategy for a given application, are the judgment on result acceptability, and the suspension of service delivery on error occurrence.

Figure 3.3 summarizes the main sources of overhead, from both structural and operational time viewpoints, involved in software fault tolerance, for tolerating one fault. In this table, the overheads caused by tests local to each variant such as input range checking, grossly wrong results, etc. are not mentioned: they are common to all approaches (in addition, they are — or should be — present in non fault-tolerant software systems as well).

3.3 ANALYSIS OF SOFTWARE FAULT TOLERANCE

This section is aimed at analyzing the approaches to software fault tolerance according to two viewpoints: dependability and cost.

3.3.1 Dependability Analysis

3.3.1.1 CLASSES OF FAULTS, ERRORS AND FAILURES TO BE CONSIDERED

We shall adopt here the classification and terminology of [Avi85], i.e., *independent faults* and *related faults*. Related faults result either from a fault specification — common to all the variants — or from dependencies in the separate designs and implementations. Related faults manifest under the form of *similar errors*, whereas independent faults usually cause *distinct*

² Although sequential execution of the variants in N -version programming can be theoretically considered, practical considerations of performance make this possibility as not very likely, except for experiments.

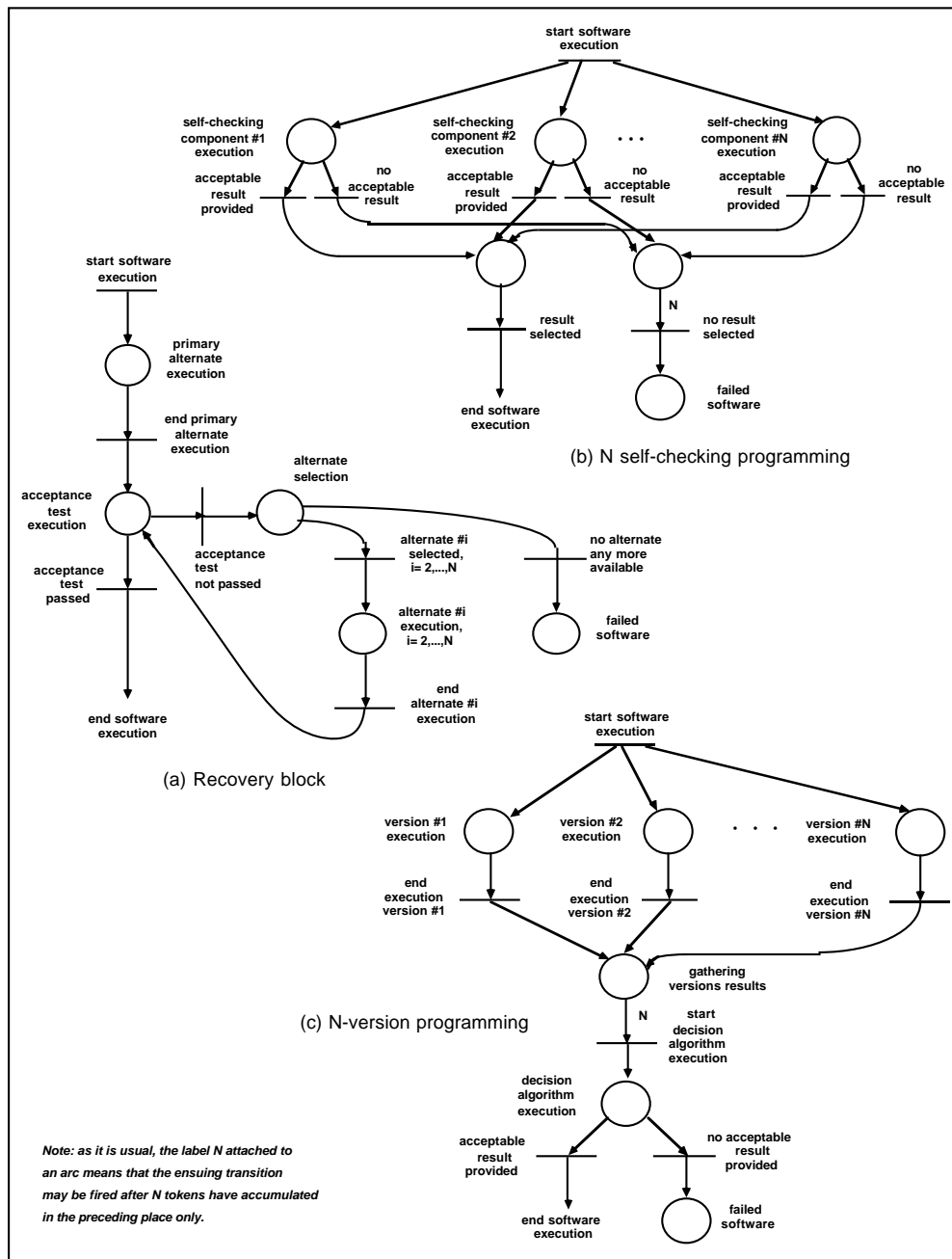


Figure 3.1 Execution models of the software-fault tolerance approaches

Method Name	Error Processing Technique		Judgement on Result Acceptability	Variant Execution Scheme	Consistency of Input Data	Suspension of Service Delivery during Error Processing	Number of Variants for Tolerance of f Sequential Faults
Recovery Blocks (RB)	Error Detection by Acceptance Test and Backward Recovery		Absolute, with respect to Specification	Sequential	Implicit, from Backward Recovery Principle	Yes, Duration Necessary for Executing One or More Variants	f+1
N Self-Checking Programming (NSCP)	Error Detection and Result Switching	Error Detection by Acceptance Test(s) Error Detection by Comparison		Parallel	Explicit, by Dedicated Mechanisms	Yes, Duration Necessary for Result Switching	
N-Version Programming (NVP)	Vote		Relative, on Variants Results				No

Figure 3.2 Main characteristics of the software-fault tolerance approaches

Method Name	Structural Overhead			Operational Time Overhead		
	Diversified Software Layer	Mechanisms (Layers Supporting the Diversified Software Layer)	Systematic		On Error Occurrence	
			Decider	Variants Execution		
Recovery Blocks (RB)	One variant and One Acceptance Test	Recovery Cache	Acceptance Test Execution	Accesses to Recovery Cache	One Variant and Acceptance Test Execution	
N Self-Checking Programming (NSCP)	Error Detection by Acceptance Tests	One variant and Two Acceptance Tests		Result Switching	Input Data Consistency and Variants Execution Synchronisation	Possible Result Switching
	Error Detection by Comparison	Three Variants	Comparators and Result Switching			
N-Version Programming (NVP)	Two Variants	Voters	Vote Execution		Usually neglectable	

Figure 3.3 Overheads involved in software-fault tolerance for tolerating one fault (with respect to non fault-tolerant software)

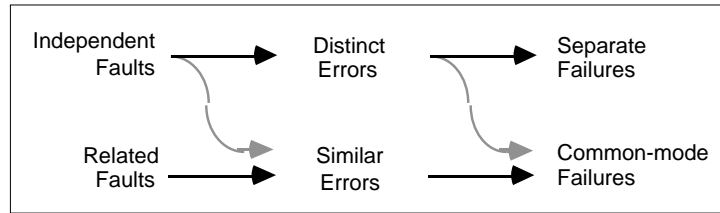


Figure 3.4 Classes of faults, errors and failures

errors, although it may happen [Avi84] that independent faults lead to similar errors. Similar errors lead to *common-mode failures*, and distinct errors usually cause *separate failures*. These definitions are illustrated by Figure 3.4.

Emphasis is put on the distinction between the different sources of failures: independent and related faults in variants and decider. In the analysis, it is assumed that only one type of fault(s), either independent or related, can be activated at each execution and produce error(s). In addition, the fault tolerance underlying mechanisms, i.e., a) recovery point establishment and restoration for the RB, and b) synchronization of the versions, cross check-points establishment and the decision mechanisms for NSCP and NVP, are not accounted for.

The failures will be classified according to two different viewpoints:

1. with respect to the type of faults whose activation has led to failure:
 - *separate failures*, which result from the activation of independent faults in the variants,
 - *common-mode failures*, which may result from the activation of either related faults or from independent faults in the decider; we shall distinguish among two types of related faults: a) related faults among the variants, and b) related faults between the variants and the decider.
2. with respect to the detection of inability to deliver acceptable results:
 - *detected failures*, when no acceptable result is identified by the decider and no output result is delivered,
 - *undetected failures*, when erroneous results are delivered.

It is assumed in the sequel that the probability of fault activation is identical for all the variants of a given architecture. This assumption is made to simplify the notation and do not alter the significance of the obtained results (the generalization to the case where the characteristics of the variants are distinguished can be easily deduced).

3.3.1.2 ANALYSIS OF RECOVERY BLOCK BEHAVIOR

Figure 3.5 presents the related- or independent-fault combinations that may affect the RB architecture made up of two alternates — a primary (P) and a secondary (S) — and of an acceptance test (AT). The table also gives the associated probabilities of fault activation and the resulting type of failures (either, detected or undetected) when the faults are not tolerated. For the sake of conciseness, when several permutations of the faulty software components correspond to an identical fault characterization, only one fault combination is shown, which is identified by symbol "◇" attached to the fault combination number, and the symbol "()" indicates the variants included in the permutations. For example, fault combination

#	Fault Combination			Characterization and Probability	Failure	
	P	S	AT			
1 [◇]	(i)	()		Independent fault in one variant: $2 q_{I, RB} (1 - q_{I, RB})$	—	—
2	i	i		Independent faults in the two variants: $(q_{I, RB})^2$	S	D
3 [◇]	(*)	(*)	i	Independent fault in the AT: $q_{ID, RB}$	CM	D
4	r	r		Related fault between the two variants: $q_{2V, RB}$	CM	D
5 [◇]	(r)	()	r	Related fault between the variants and the AT: $q_{RVD, RB}$	CM	U
6	r	r	r			

Faults: : no fault, i : independent, * : no fault or independent, r : related

#[◇]: several permutations, () : variants included in the permutations

Failures: S: Separate, CM: Common-Mode; D: Detected, U: Undetected

Figure 3.5 Recovery blocks fault-failure analysis

1[◇] of Figure 3.5 covers two permutations, each having the same probability of activation: $q_{I, RB} (1 - q_{I, RB})$, where $q_{I, RB}$ denotes the probability of activation of an independent fault in one of the variants of the recovery blocks. Furthermore, fault combination 1[◇] is tolerated and thus does not lead to the failure of the RB architecture.

It is assumed that related faults between the variants manifest under the form of similar errors. However, we assume that independent faults in the variants manifest under the form of distinct errors³. In particular, this results in the fact that the activation of two or more independent faults is either tolerated (fault combination 2) or detected (fault combinations 3[◇], 4, 5[◇] and 6).

Fault combination 3[◇] covers all the cases when an independent fault is activated in the decider of the recovery blocks: the acceptance test; all the cases are grouped in the probability denoted by $q_{ID, RB}$. It is assumed that the AT behaves consistently with respect to each variant (no error compensation), thus when an independent fault is activated in the AT, a detected failure occurs, since undetected failures characterized by the activation of a fault in the AT are the consequences of the activation of a related fault between the variants and the decider (fault combinations 5[◇] and 6).

In the case of fault combination 4, the associated probability of activation, denoted $q_{RV, RB}$, corresponds to a related fault between the two variants of the RB.

Probability $q_{RVD, RB}$ covers all the cases of activation of a related fault between the variants and the decider (fault combinations 5[◇] and 6). This corresponds to the only fault combinations that lead to an undetected failure for RB.

³ Although they could be traced to independent faults, faults leading to similar errors are not distinguishable at the execution level from related faults and thus they will be merged into the category of related faults.

3.3.1.3 ANALYSIS OF N SELF-CHECKING PROGRAMMING BEHAVIOR

The NSCP architecture is made up of 2 self-checking components (SCC), each of them being made up of 2 variants and a comparator C. Although all 4 variants V_i , $i = 1, \dots, 4$, are executed simultaneously, only one SCC is *acting*, i.e., actually delivering output results, the other being used as a spare. The two probabilities of fault activation for the two comparators are assumed to be identical.

Figure 3.6 presents the related- or independent-fault combinations that may affect the NSCP architecture, their associated probabilities of activation and the resulting type of failure, if they are not tolerated. For sake of conciseness, the comparators have been omitted in the description of the fault combinations they are not involved in. The presentation of the fault combinations and the notation used for the associated probability of activation are analogous to the ones used for RB; in the sequel we focus essentially on discussing some points that are specific to the NSCP architecture.

As the comparators are assumed to be identical from the failure probability viewpoint, a) the activation of an independent fault in the comparator is fully characterized by the fault combination 6, which leads to a detected failure, and b) the only case of related fault between the variants and the comparators to be considered is identified by fault combination 12.

It is important to note that, although they correspond to the activation of a related fault between two variants, the fault combinations 7 and 9[◇] are respectively, tolerated and detected, by the NSCP architecture. All other related-fault combinations result in undetected failures.

3.3.1.4 ANALYSIS OF N-VERSION PROGRAMMING BEHAVIOR

Figure 3.7 presents the related- or independent-fault combinations that may affect the NVP architecture.

In this case, it appears that all multiple independent fault combinations and an independent fault in the decider are detected (fault combinations 2[◇] to 4), and that all related fault combinations lead to undetected failures (fault combinations 5[◇] to 7).

3.3.1.5 CHARACTERIZATION OF THE PROBABILITIES OF FAILURE

Figure 3.8 gives the respective contribution of the separate/common-mode and detected/undetected failure viewpoints to the probability of failure of the considered software fault-tolerant architectures.

As could be expected, the table shows that, although the detected failures may correspond to both separate failures or common mode failures, the undetected failures correspond exclusively to common mode failures.

The comparison of the various failure probabilities of Figure 3.8 is difficult due to the difference in the values of the parameters characterizing each architecture. However, it is possible to make some interesting general remarks.

Although a large number of experiments have been carried out to analyze NVP, no quantitative study has been reported on the reliability associated to the decider, except for some qualitative remarks concerning a) the granularity of the decision with respect to the variables checked [Tso87], b) the magnitude of the allowed discrepancy [Kel88], and c) the investigation of the behavior of various voter schemes reported in [Lor89]. However, it has to be noted that the probabilities associated to the deciders can be significantly different. Due to the gener-

Fault Combination acting SCC spare SCC		Characterization and Probability	Failure	
#	$\begin{matrix} \boxed{V1} & \boxed{V2} & \boxed{C} \\ \boxed{V3} & \boxed{V4} & \boxed{C} \end{matrix}$			
1 [◊]	$\begin{matrix} \boxed{(i)} & \boxed{()} & \boxed{} \\ \boxed{()} & \boxed{()} & \boxed{} \end{matrix}$	Independent fault in one variant: $4 q_{I,NSCP} (1 - q_{I,NSCP})^3$	—	—
2 [◊]	$\begin{matrix} \boxed{(i)} & \boxed{(i)} & \boxed{} \\ \boxed{()} & \boxed{()} & \boxed{} \end{matrix}$	Indep. faults in two variants belonging, either to the acting or to the spare SCCs: $2 (q_{I,NSCP})^2 (1 - q_{I,NSCP})^2$	—	—
3 [◊]	$\begin{matrix} \boxed{(i)} & \boxed{()} & \boxed{} \\ \boxed{(i)} & \boxed{()} & \boxed{} \end{matrix}$	Independent faults in two variants belonging to distinct SCCs: $4 (q_{I,NSCP})^2 (1 - q_{I,NSCP})^2$	S	D
4 [◊]	$\begin{matrix} \boxed{(i)} & \boxed{(i)} & \boxed{} \\ \boxed{(i)} & \boxed{()} & \boxed{} \end{matrix}$	Ind. fault in three variants: $4 (q_{I,NSCP})^3 (1 - q_{I,NSCP})$	S	D
5	$\begin{matrix} \boxed{i} & \boxed{i} & \boxed{} \\ \boxed{i} & \boxed{i} & \boxed{} \end{matrix}$	Independent faults in four variants: $(q_{I,NSCP})^4$	S	D
6	$\begin{matrix} \boxed{*} & \boxed{*} & \boxed{i} \\ \boxed{*} & \boxed{*} & \boxed{i} \end{matrix}$	Independent fault in the comparator: $q_{ID,NSCP}$	CM	D
7	$\begin{matrix} \boxed{} & \boxed{} & \boxed{} \\ \boxed{r} & \boxed{r} & \boxed{} \end{matrix}$	Related fault between the variants belonging to the spare SCC: $q_{2V,NSCP}$	—	—
8	$\begin{matrix} \boxed{r} & \boxed{r} & \boxed{} \\ \boxed{} & \boxed{} & \boxed{} \end{matrix}$	Related fault between the variants belonging to the acting SCC: $q_{2V,NSCP}$	CM	U
9 [◊]	$\begin{matrix} \boxed{(r)} & \boxed{()} & \boxed{} \\ \boxed{(r)} & \boxed{()} & \boxed{} \end{matrix}$	Related fault between the variants belonging to distinct SCCs: $4 q_{2V,NSCP}$	CM	D
10 [◊]	$\begin{matrix} \boxed{(r)} & \boxed{(r)} & \boxed{} \\ \boxed{(r)} & \boxed{()} & \boxed{} \end{matrix}$	Related fault between three variants: $4 q_{3V,NSCP}$	CM	U
11	$\begin{matrix} \boxed{r} & \boxed{r} & \boxed{} \\ \boxed{r} & \boxed{r} & \boxed{} \end{matrix}$	Related fault between four variants: $q_{4V,NSCP}$	CM	U
12	$\begin{matrix} \boxed{r} & \boxed{r} & \boxed{r} \\ \boxed{r} & \boxed{r} & \boxed{r} \end{matrix}$	Related fault between the four variants and the comparator: $q_{RVD,NSCP}$	CM	U

Faults: $\boxed{}$: no fault, \boxed{i} : independent, $\boxed{*}$: no fault or independent, \boxed{r} : related

#[◊]: several permutations, $\boxed{()}$: variants included in the permutations

Failures: S: Separate, CM: Common-Mode; D: Detected, U: Undetected

Figure 3.6 N self-checking programming fault-failure analysis

Fault Combination		Characterization and Probability	Failure	
#	V1 V2 V3 v			
1 [∅]	(i) () ()	Independent fault in one variant: $q_{I,NVP} (1 - q_{I,NVP})^2$	—	—
2 [∅]	(i) (i) ()	Independent faults in two variants: $3 (q_{I,NVP})^2 (1 - q_{I,NVP})$	S	D
3	i i i	Independent faults in the three variants: $(q_{I,NVP})^3$	S	D
4	* * * i	Independent fault in the voter: $q_{ID,NVP}$	CM	D
5 [∅]	(i) (i) ()	Related fault between two variants: $3 q_{2V,NVP}$	CM	U
6	r r r	Related fault between the three variants: $q_{3V,NVP}$	CM	U
7	r r r r	Related fault between the three variants and the voter: $q_{RVD,NVP}$	CM	U

Faults: : no fault, i : independent, * : no fault or independent, r : related

#[∅]: several permutations, () : variants included in the permutations

Failures: S: Separate, CM: Common-Mode; D: Detected, U: Undetected

Figure 3.7 *N*-version programming fault-failure analysis

Arch. "X"	Probability of Failure: $q_{S,X} = q_{S,D,X} + q_{S,U,X}$	
	Detected Failure: $q_{S,D,X}$	Undetected Failure: $q_{S,U,X}$
RB	S: $(q_{I,RB})^2$ CM: $q_{ID,RB} + q_{2V,RB}$	CM: $q_{RVD,RB}$
NSCP	S: $4 (q_{I,NSCP})^2 [1 - q_{I,NSCP} + \frac{q_{I,NSCP}}{4}]$ CM: $q_{ID,NSCP} + 4 q_{2V,NSCP}$	CM: $q_{2V,NSCP} + 4 q_{3V,NSCP} + q_{4V,NSCP} + q_{RVD,NSCP}$
NVP	S: $3 (q_{I,NVP})^2 [1 - \frac{2}{3} q_{I,NVP}]$ CM: $q_{ID,NVP}$	CM: $3 q_{2V,NVP} + q_{3V,NVP} + q_{RVD,NVP}$

Figure 3.8 Detected/undetected failures and separate/common-mode failures

ality and the simplicity of the functions performed by the deciders of the NSCP (comparison) and NVP (voting) architectures, it is likely that these probabilities be ranked as follows:

$$q_{ID,NSCP} \leq q_{ID,NVP} \ll q_{ID,RB} \quad \text{and} \quad q_{RVD,NSCP} \leq q_{RVD,NVP} \ll q_{RVD,RB} \quad (3.1)$$

In the case of separate failures, the influence of the activation of independent faults differs significantly: for RB, the probability of separate failure is equal to the square of the probability of activation of independent faults, whereas it is almost the triple for NVP and the quadruple for NSCP⁴. This difference results from the number of variants and the type of decision used.

The fact that all related faults in NVP result in undetected failures does not mean that RB and NSCP are the only architectures that enable some related-fault combinations to be detected. Such a singularity is due to the limitation to the analysis of software redundancy for tolerating one fault⁵.

Each entry of the table of Figure 3.8 summarizes the probabilities of detected/undetected failure of the software fault-tolerant architectures considered:

- $q_{S,D,X} = P \{ \text{detected failure of the software fault-tolerant architecture X — execution} \}$,
- $q_{S,U,X} = P \{ \text{undetected failure of the software fault-tolerant architecture X — execution} \}$,
- $q_{S,X} = q_{S,D,X} + q_{S,U,X} = P \{ \text{failure of the software fault-tolerant architecture X — execution} \}$.

Related faults between the variants has no impact on the probability of undetected failure of the RB architecture, whereas, it is the major contribution for NSCP and NVP. However, the comparison of the respective probabilities of undetected failures is not an easy task.

The results obtained agree with what has previously been reported in the literature, i.e., that the reliability of a fault-tolerant software is dominated by related faults (see e.g., [Kni86, Eck91]). This is hardly unexpected for anyone who is familiar with fault tolerance: although the underlying phenomena are different, the limit due to related faults can be compared to the usual limit in reliability for hardware-fault tolerance brought about by coverage deficiencies (either from a theoretical [Bou69, Arn73] or from a practical [Toy78] viewpoint). However, the results show that design diversity indeed brings in a significant reliability improvement as the impact of independent faults which are the major source of failure in traditional — non fault-tolerant — software, become negligible. In addition, it has been shown experimentally that some specification faults are in fact detected during the execution of multi-variant software (see e.g., [Bis86, Tra88]), thus reducing the proportion of related faults leading to failure in a multi-variant software.

3.3.2 Cost Analysis

This section is aimed at giving an estimation of the additional cost introduced by fault tolerance. Since design diversity obviously affects costs differently according to the life-cycle phases, the starting point is the cost distribution among the various activities of the life-cycle for a classical, non fault-tolerant, software. This is provided by Figure 3.9, where:

⁴ Due to the usually low value of the $q_{I,X}$ factors, the influence of separate failures is practically limited to the consideration of the $(q_{I,X})^2$ terms.

⁵ It is worth noting that increasing the number of versions would also enable some related faults to be detected by NVP (e.g., see Section 3.4.5.4).

Activities			Life-Cycle Cost Breakdown [Zel 79]	Maintenance Cost Breakdown [Ram 84]			Multipliers for Critical Applications [Boe 81]	Cost Distribution	
				Mnemonic	Corrective	Adaptive		Perfective	Development
Development	Requirements	R	3 %			55 %	1.3	8 %	6 %
	Specification	S	3 %				1.3	8 %	7 %
	Design	D	5 %	20 %	25 %		1.3	13 %	14 %
	Implementation	I	7 %				1.3	19 %	19 %
	V & V	V	15 %				1.8	52 %	54 %
Maintenance			67 %	100 %				100 %	100 %
			100 %						

Figure 3.9 Software cost elements for non fault-tolerant software

- a) The life-cycle model that we have adopted is a simplified one [Ram84], an interesting feature of which with respect to our purpose is that all the activities relating to verification and validation (V&V) are grouped separately.
- b) The three categories of maintenance are defined according to [Swa76], and cover all the operational life:
 - corrective maintenance concerns fault removal; it thus involves design, implementation and testing types of activities;
 - adaptive maintenance adjusts software to environmental changes; it thus involves also specification activity;
 - perfective maintenance is aimed at improving the software's function; it thus actually concerns software's evolution, and as such involves activities similar to all of the activities of the development, starting from modified requirements.
- c) The cost breakdown for the life-cycle [Zel79] and for maintenance [Ram84] are general, and do not address a specific class of software; however, since we are concerned with critical applications, some multiplicative factors have to be accounted for, which depend upon the considered activity [Boe81].
- d) The last two columns, which have been derived from the data displayed in the other columns, give the cost distribution among the activities of the life-cycle in two cases: development only, development and maintenance.

From this table, it appears that maintenance does not affect significantly the cost *distribution* over the other activities of the life-cycle (the discrepancy is likely to be lower than the accuracy of the obtained figures); so, considering in the following one case only, say the development case as it refers to the *production* of software, is general, and covers also the whole life-cycle as we are concerned with relative costs.

To determine the cost of a fault-tolerant software, it is necessary to introduce factors a) enabling the overheads associated with the decision points and the decider(s) to be accounted for, as well as b) enabling the cost reduction in V&V induced by the commonalities among variants to be accounted for. These commonalities include actual V&V activities, such as back-to-back testing, and V&V tools, such as test harnesses. An accurate estimation of such

Number of faults tolerated	Fault Tolerance Method		N	$\left(\frac{C_{FT}}{C_{NFT}}\right)_{\min}$	$\left(\frac{C_{FT}}{C_{NFT}}\right)_{\max}$	$\left(\frac{C_{FT}}{C_{NFT}}\right)_{\text{av}}$	$\left(\frac{C_{FT}}{N C_{NFT}}\right)_{\text{av}}$
1	Recovery Blocks		2	1.33	2.17	1.75	.88
	N Self-Checking Programming	Acceptance Test					
		Comparison	4	2.24	3.77	3.01	.75
	N-Version Programming		3	1.78	2.71	2.25	.75
2	Recovery Blocks		3	1.78	2.96	2.37	.79
	N Self-Checking Programming	Acceptance Test					
		Comparison	6	3.71	5.54	4.63	.77
	N-Version Programming		4	2.24	3.77	3.01	.75

Figure 3.10 Cost of fault-tolerant software vs. non fault-tolerant software

factors is out of reach given the current state of the art; we however think that it is possible to give reasonable ranges of variations. We introduce the following factors:

- r: multiplier associated with the decision points, with $1 \leq r \leq 1.2$;
- s: multiplier associated with the decider, with $1 \leq s \leq 1.1$ for NVP and NSCP when error detection is performed through comparison, and $1 \leq s \leq 1.3$ for RB and NSCP when error detection is performed through acceptance tests; this difference is aimed at reflecting the different natures of the decider, i.e., the fact that it is specific in the case of acceptance test, whereas it is generic in the cases of comparison and of vote;
- u: proportion of the testing activities which are performed once for all variants (e.g., provision for test environments and harnesses), with $0.2 \leq u \leq 0.5$;
- v: proportion of the testing activities of each variant which take advantage of the existence of several variants (e.g., back-to-back testing), with $0.3 \leq v \leq 0.6$;
- w: cost reduction factor for the testing activities performed in common for several variants, with $0.2 \leq w \leq 0.8$.

The cost of a fault-tolerant software (C_{FT}) with respect to the cost of a non fault-tolerant software (C_{NFT}) is then given by the following expression:

$$C_{FT}/C_{NFT} = R + rsS + [Nr + (s - 1)](D + I) + r\{us + (1 - u)N[vw + (1 - v)]\}V$$

where $R, S, D, I,$ and V refer to the life-cycle activities of Figure 3.9, and N is the number of variants.

Figure 3.10 gives the ranges for the ratio C_{FT} / C_{NFT} , as well as the average values and the average values per variant. In this table, we do not make a distinction between RB and NSCP with error detection by acceptance test: their respective differences with respect to the cost issues are likely to be masked by our abstract cost model.

The results displayed in Figure 3.10 enable the quantification of the usual qualitative statement according to which an N variant software is less costly than N times a non fault-tolerant software. In addition, the figures that appeared previously in the literature fall within the ranges displayed in the table:

- the overhead for RB, for 2 variants, was estimated at 60% for the experiment conducted at the University of Newcastle upon Tyne [And85], that is a ratio C_{FT}/C_{NFT} equal to 1.6,
- the cost of NVP, for 3 variants, in the PODS project [Bis86] was estimated to 2.26 times the cost of a single variant program.

3.4 DEFINITION AND ANALYSIS OF HARDWARE-AND-SOFTWARE FAULT TOLERANT ARCHITECTURES

This section presents examples of architectures providing tolerance to both hardware and software faults. Emphasis will be put on: a) the dependencies among the software and hardware fault tolerance and b) the impact of the solid and soft character of the software faults, related to their persistence with respect to the computation process and their recoverability, in the definition of the architectures.

Two levels of fault tolerance requirements are successively investigated: a) architectures tolerating a single fault, and b) architectures tolerating two consecutive faults⁶. The architectures will be identified by means of a condensed expression of the form: $X/i/j/...$, where label X stands for the acronym of the software fault tolerance method, $X \in \text{RB, NSCP, NVP}$, i is the number of hardware faults tolerated and j is the number of software faults tolerated. Further labels will be added to this expression when necessary.

Due to the scope of this chapter, the architectures are described from a deliberately highly abstracted view. As a consequence, distinguishing features, such as a) the overhead involved by inter-component communication for synchronization, decision-making, data consistency, etc., and b) the differences in memory space necessary for each architecture, are not addressed. More specific attempts to the definition of hardware-and-software fault-tolerant architectures have already appeared in the literature, e.g., the work reported in [Kim84, Kim89] which extends the recovery block approach, or in [Lal88] which is based on N -version programming.

3.4.1 Persistence and Recoverability of Software Faults

Consideration of the persistence criterion leads to distinguish *solid* and *soft* faults. Such a distinction is usual in hardware, where solid faults are usually synonymous to permanent faults, and soft faults are usually synonymous to temporary faults, either transient or intermittent. The solidity or softness character of a fault plays an important role in fault tolerance: a component affected by a permanent fault has to be made passive after fault manifestation, whereas a component affected by a soft fault can be used in subsequent executions after error recovery has taken place.

Let us now consider software faults of *operational* programs, i.e., programs that have been thoroughly debugged, where the remaining sources of problems are likely to be more fault *conditions* rather than solid faults: limit conditions, race conditions, or strange underlying hardware conditions. Because of a slight change in the execution context, the corresponding fault conditions may not be gathered, and the software may not fail again. The notion of temporary fault may thus be extended to software, and it has been noted in practice: it has been introduced in [Elm72], has been devoted much attention in [Gra86] and is being used as a basis for increasing diversity among several variants of the same software through the

⁶ These requirements can be related respectively, to the classical Fail Op/Fail Safe and Fail Op/Fail Op/Fail Safe requirements used in the aerospace community for the hardware fault tolerance.

concept of "data diversity" (see, e.g., [Amm87, Chr94]). A temporary software fault may then be characterized as a fault whose likelihood of manifestation recurrence upon re-execution is low enough to be neglected.

Another, important, consideration to account for is the notion of local and global variables for the components, in relation with error recovery actions (in the broad sense of the term, including the actions to be performed so that a failed variant in NSCP and in NVP may take part to future executions [Tso86]). Let us call a *diversity unit* the program comprised between two decision points. As a general rule, it can be said that recovery requires that the diversity units have to be procedures (in the sense that their activation and behavior do not depend on any internal permanent data). Stated in other terms, all the data necessary for the processing tasks of a diversity unit have to be global data. The globality of the data for a diversity unit may:

- originate directly from the nature of the application; an example is provided by the monitoring of a physical process (e.g., nuclear reactor protection [Gme79]) where tasks are initialized by acquisition of sensor data, and do not utilize data produced in previous processing steps;
- result from the system partitioning, either naturally, or from the transformation of local data into global data, at the expense in the latter case of an overhead and of a decrease in diversity (the specification of the decision points has to be made more precise); a — deliberately over-simplified — example is provided by a filtering function which would constitute a diversity unit: the past samples should be made part of the global data.

The preceding discussion applies to all the software fault tolerance methods. It is however noteworthy that some arrangements can be made to this general rule in some specific, application-dependent, cases. Examples are as follows:

- if the above-mentioned overhead cannot be afforded, or if it is estimated that transforming local data into global data is a too severe decrease in diversity, an alternative solution exists for NSCP and NVP: eliminate from further processing a failed variant;
- the application of state estimation techniques issued from the control theory (see e.g., [Cag85]).

As a summary, we shall adopt in the sequel the following definitions for soft and solid faults:

- a *soft* software fault is a fault which is temporary and recoverable,
- a *solid* software fault is a fault which is permanent or which cannot be recovered.

Finally, we implicitly considered that decision points were recovery points. In NSCP and NVP, a distinction in the decision points can be made between those which are cross-check points and those which are recovery points, the latter being possibly in a smaller number than the former [Tso86].

3.4.2 Design Decisions when Implementing Design Diversity

Among the many issues that have to be dealt with design diversity [Avi85, Avi86], two of them are especially important — and correlated — *with respect to architectural considerations*: a) the number of variants, and b) the level of fault tolerance application.

Independently from any economic consideration (see Section 3.3.2), the *number of variants* to be produced for a given software fault tolerance method is directly related to the number of faults to be tolerated (see Figure 3.2). It will be seen in this section that the soft or solid character of the software faults has a significant impact on the architecture only when dealing with the tolerance of more than one fault. It has also to be reminded that an architecture tolerating a solid fault is also able to tolerate a (theoretically) infinite sequence of soft faults — provided there is no fault coincidence phenomenon. It is noteworthy that the relation between the likelihood of such faults and the number of variants is not simple: whether increasing the number of variants will increase or decrease the number of related faults depends on several factors, some of them being antagonistic [Avi84, Kni86]. However, in NVP, an incentive to increase the number of variants is that two similar errors can outvote a good result in a three-version scheme, whereas they would be detected in a four-version scheme, thus providing an increase in safety (situations contrary to safety being defined as corresponding to an undetected error).

The *level of application of fault tolerance* encompasses two aspects: a) at what level of detail should be performed the decomposition of the system into components that will be diversified?, and b) which layers (application software, executive, hardware) have to be diversified?

The answer to the first question should at first sight result from a trade-off between two opposite considerations: on one hand, small size components enable a better mastering of the decision algorithms; on the other hand, large size components favor diversity. In addition, the decision points are "non diversity" points (and synchronization points for NSCP and NVP); as such, they have to be limited: decision points are *a priori* necessary for the interactions with environment only (sensor data acquisition, delivery of orders to actuators, interactions with operators, etc.). However, additional compromises may result from performance considerations.

Concerning the *layers where to apply diversity*, the methods mentioned can be applied to any of the software layers, either of application software, or of the executive software. They can also be applied to the hardware layer(s) [Mar82, Rou86]. The states — with respect to the computation process — of distinct variants are different. A consequence is that when the variants are executed in parallel (NSCP and NVP), thus on distinct (redundant) hardware, diversity of a given layer leads to states of the underlying layers that are different, even if they are not diversified — except of course at the decision points. A decision concerning whether layers underlying the application software have to be diversified, should include additional considerations, such as determining the influence of the portions of executive software and of hardware that are specifically designed for the considered application, and what confidence to place on experience validation for those parts which are off-the-shelf.

3.4.3 Structuring Principles for the Definition of Architectures

Structuring is a prerequisite to the mastering of complexity. This is especially true when dealing with fault tolerance [And81, Ran84, Neu86].

A usual, and useful, principle when dealing with hardware-fault tolerance is that the fault tolerance mechanisms should go along, and respect, the structuring of a system into layers [Sie82]. Especially, it is desirable that each layer be provided with fault tolerance mechanisms aimed at processing the errors produced in the considered layer, with respect a) to performance considerations in terms of time to recover from an error and b) to the damages created by error propagation. Implementation of this principle at the hardware layer when dealing with

software fault tolerance requires that the redundant hardware components are in the same state with respect to the computation process in the absence of error. Such a condition can obviously be satisfied only if the variants are executed sequentially, i.e., in the RB approach.

However, the diagnosis of hardware faults may be made possible by taking advantage of the syndromes provided by the decider(s) of the considered software fault tolerance method considered.

Another useful structuring mechanism is the notion of *error confinement area* [Sie82]. This notion cannot be separated from the architectural elements considered. In relation with the scope of this chapter, the architectural elements considered will be:

- the elements providing the services necessary for an application software to be executed, i.e., hardware and the associated executive software; such elements will be termed — abusively, in the sake of conciseness — as *hardware components*;
- the *variants* of the application software.

Consideration of both hardware and software faults leads to distinguish hardware error confinement areas (HECAs) and software error confinement areas (SECAs). In our case, a HECA will cover at least one hardware component and a SECA will cover at least one software variant. It is noteworthy that, due to our definition of hardware component, a HECA corresponds to the part of the architecture that is made passive after occurrence of a solid hardware fault, and can thus be interpreted as a Line Replaceable Unit (LRU).

3.4.4 Architectures Tolerating a Single Fault

Three architectures are considered that correspond to the implementation of the three software fault-tolerance methods of Section 3.2. Figure 3.11 illustrates the configurations of the SECAs and HECAs for each case. The intersections between the SECAs and the HECAs characterize the software and hardware fault tolerance dependencies of the architectures.

3.4.4.1 THE RB/1/1 ARCHITECTURE

This architecture is obtained by the duplication of the RB composed of two variants on two hardware components. Two variants and their associated instances of the acceptance test constitute two distinct SECAs and intersect each HECA. Accordingly, each hardware component is software fault tolerant due to the RB method: the independent faults in a variant are tolerated, while the related faults between the variants are detected; however related faults between each variant and the acceptance test cannot be tolerated neither detected.

The hardware components operate in hot standby redundancy and execute at each time the same variant. This allows the design of a high coverage concurrent comparison between the results of the acceptance tests and the results provided by the hardware components to handle specifically the hardware faults. It is noteworthy that when a discrepancy between the results of the hardware components is detected during the execution of the primary alternate or of the acceptance test, it might be interesting to call for the execution of the secondary to tolerate the fault (should the fault be a soft fault). If the discrepancy would persist (what would be the case upon manifestation of a solid fault), the failed HECA could be identified by running diagnosis programs on each HECA, thus, the continuity of service would be ensured and the failed HECA made passive.

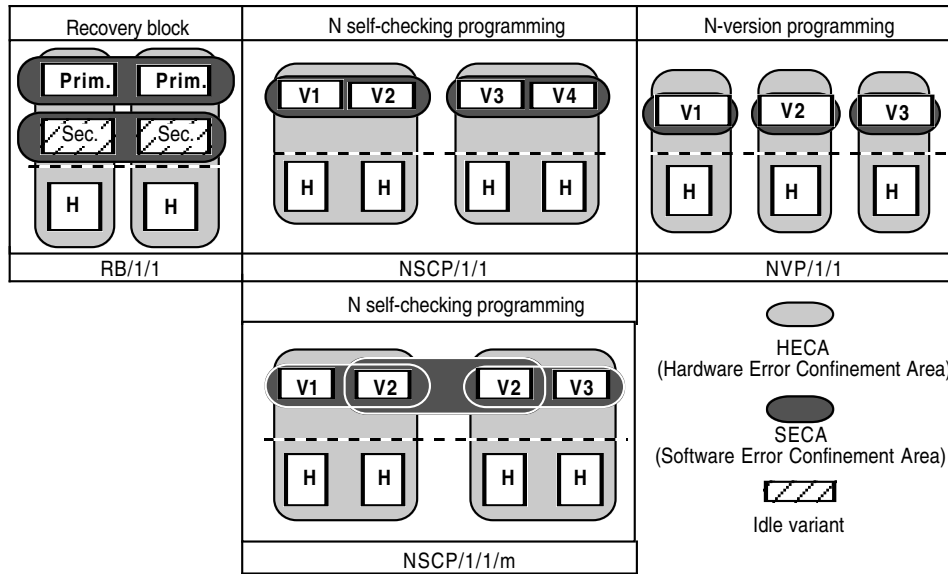


Figure 3.11 Architectures tolerating a single fault

It is worth noting that after this hardware degradation, a) the architecture is still software fault tolerant, and b) the subsequent hardware faults can be detected either by means of the acceptance test or by the periodic execution of the diagnosis.

3.4.4.2 EXAMPLES OF NSCP/1/1 ARCHITECTURES

The basic architecture (NSCP/1/1 in Figure 3.11) is made up of:

- four hardware components grouped in two pairs in hot standby redundancy, each pair of hardware components forming a HECA;
- four variants grouped in two pairs; each pair constitutes a software self-checking component, error detection being carried out by comparison; each pair forms a SECA.

Each SECA is associated to a HECA. The comparison of the computational states of the hardware components cannot be directly performed due to the diversification imposed by the variants; however, the comparison between the results of each pair of variants covers also the two hardware components composing each HECA with respect to the hardware faults (including design faults); a HECA is thus also a hardware self-checking component.

In case of discrepancy between the results provided by the paired variants in one HECA, irrespective of the type of fault, the associated HECA is possibly switched out and the results are delivered by the other HECA. Should the discrepancy occurs repeatedly, characterizing thus, a solid hardware fault, then the designated HECA would be made passive. The degraded structure after this passivation still enables both software and hardware faults to be detected.

Besides the nominal tolerance of an independent software fault, the architecture ensures supplementary tolerance and detection features: a) the tolerance of two simultaneous independent faults in a SECA, b) the detection of a related fault among two variants (each per-

taining to one of the two disjoint SECAs) and c) the detection of three or four simultaneous independent software faults.

The NSCP/1/1 architecture corresponds to the principle of the architecture implemented in the Airbus A320 [Rou86]. However, in some applications, the requirement of four variants would be prohibitive; it is worth noting that a modified architecture (NSCP/1/1/m) can be obtained, based on three variants only (Figure 3.11).

The major difference in error processing between the NSCP/1/1 and NCSP/1/1/m architectures can be identified when considering the activation of a software fault in V2. It is important to note that this would result in a discrepancy in both self-checking components, thus implying an associated SECA covering all the four software components and preventing any software fault tolerance. As this is the only event that can lead to such a syndrome (under the hypothesis of single independent fault), the "correct" result is immediately available as the one provided by V1 or V3, hence, the SECA associated to V2 on Figure 3.11. However, as a consequence, all the fault tolerance and detection capabilities of the NSCP/1/1 architecture termed above "supplementary" are lost.

3.4.4.3 THE NVP/1/1 ARCHITECTURE

The NVP/1/1 architecture is a direct implementation of the NVP method. It is made up of three hardware components, each running a distinct variant. The handling of the hardware faults (including design faults) and of the software faults is common and performed at the software layer by the decider of the NVP method. Besides the tolerance to an independent fault in a single variant, the architecture allows the detection of independent faults in two or three variants.

The problem of discrimination between hardware and software faults, so that the passivation of a hardware component occurs only upon occurrence of a solid fault, gives an example of the dependency between software and hardware fault tolerance. Due to the soft character considered for the software faults, a diagnosis of (solid) hardware fault could be easily implemented as the monitoring of a repeatedly disagreeing hardware component. After passivation of the failed hardware component, the decider has to be reconfigured as a comparator, which thus ensures a fail-safe characteristic to the architecture in case of a subsequent activation of a hardware or a software fault.

3.4.5 Architectures Tolerating Two Consecutive Faults

When dealing with the tolerance of two faults, the distinction between soft and solid software faults comes immediately into play:

- if the software faults can be considered as soft faults, then the number of variants is unchanged with respect to the architectures aimed at tolerating one fault; so these architectures will be of the type $X/2/1$, $X \in \{RB, NSCP, NVP\}$,
- if the software faults are to be considered as solid faults, then the number of variants must be increased in order to cope with the discarding of a failed variant from further execution; so the corresponding architectures will be of the type $X/2/2$.

The considered architectures for the tolerance of two faults are displayed on Figure 3.12. The first three architectures (RB/2/1, NSCP/2/1 and NVP/2/1) are characterized by the tolerance of two hardware faults and of a single software fault. The impact of the solid character

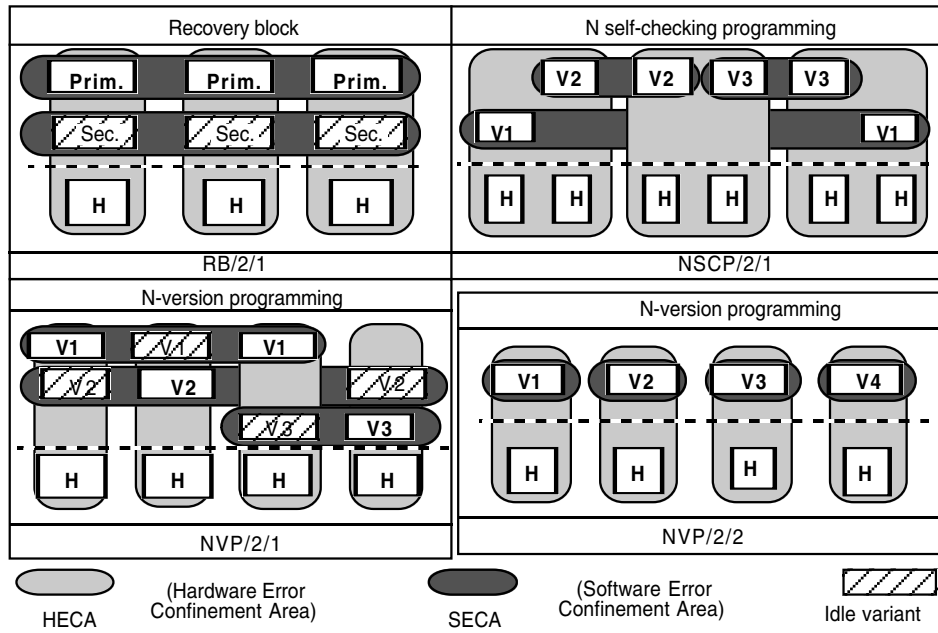


Figure 3.12 Architectures tolerating two consecutive faults

of the software faults is illustrated in the case of an architecture using the NVP method; this structure (NVP/2/2) is designed to tolerate two consecutive (solid) faults in hardware or software.

3.4.5.1 THE RB/2/1 ARCHITECTURE

This architecture is made up of three hardware components arranged in TMR. Software-fault tolerance capability is unchanged with respect to RB/1/1 architecture. Upon solid hardware fault manifestation, the architecture is degraded through the passivation of the corresponding hardware component, thus resulting in an architecture analogous to the RB/1/1 architecture.

Accordingly, even if they are basically useless in the handling of the first hardware fault, local diagnosis must be incorporated in each hardware component.

3.4.5.2 THE NSCP/2/1 ARCHITECTURE

This architecture is a direct extension of the NSCP/1/1/m architecture. A supplementary duplex HECA is added that supports a software self-checking component made up of two variants. A symmetric distribution of the three SECAs among the three HECAs is thus obtained. It is noteworthy that all the variants are duplicated; this allows an instantaneous diagnosis of the hardware faults from the syndrome obtained by comparing the results delivered by all the hardware components. The architecture allows also the detection of simultaneous independent faults in two or three variants.

3.4.5.3 THE NVP/2/1 ARCHITECTURE

The NVP/2/1 architecture is derived from the NVP/1/1 architecture by addition of a hardware component without introducing another variant. To maintain the software-fault tolerance capability after passivation of one hardware component, it is necessary that at least two instances of each variant pertain to two distinct HECAs. Thus six variant instances have to be distributed among the four HECAs. A possible configuration⁷ is the one shown in Figure 3.13.

Among the two distinct variants associated to each HECA, one is active and the other is idle. At a given step, three hardware components execute three distinct variants and the fourth hardware component executes a replica of one of the variants (V1 in this configuration). Besides the tolerance to an independent software fault, the architecture allows the detection of two or three simultaneous independent faults.

The tolerance of an independent fault can be obtained by a decision based on a vote in which the knowledge that two variants are identical is incorporated. The unbalanced numbers of executions of the variants can be used to improve the diagnosability with respect to the hardware faults by a double vote decision (each vote includes the results of the non duplicated variants and only one of the results of the duplicated variant); this is illustrated by considering the following cases:

- activation of a hardware fault in one of the hardware components executing the duplicated variant (V1),
- activation of a software fault in the duplicated variant,
- activation of a hardware fault in one of the hardware components executing the non duplicated variants (V2, V3) or of a software fault in one of these variants.

In the first case, the fault is, of course, easily tolerated and diagnosed as the obtained syndrome consists of a) an agreement among the three results in one vote and b) a discrepancy among the results of the second vote, hence, designating as false the result of the duplicated variant.

In the second case, the decider will identify a non-unanimity in both votes designating as false the results supplied by the duplicated variant; such a syndrome enables the duplicated variant to be diagnosed as failed and thus the "correct" result is immediately available as the one provided by the non duplicated ones.

In the last case, the tolerance is immediate, but the votes do not enable the fault to be diagnosed. Based on the assumption of non recurrent nature of the software faults, the diagnostic of hardware fault can result from the repeatedly failure of a hardware component. However, it is worth noting that another form of diagnosis is possible here that would allow to relax this assumption: upon occurrence of a localized fault (i.e., imputable to either one SECA or one HECA) the next execution step is performed after a reconfiguration of the active variants that matches the duplicated variant with the identified HECA; the decider will then have to solve one of the two cases identified above. A systematic rotation of the duplicated variants would also contribute to such a diagnosis.

After passivation of a failed hardware component, the active variants are reconfigured so that the SECAs be distributed among the remaining HECAs to form disjoint areas.

Figure 3.13 shows the distribution of active and idle variants among the three remaining HECAs after passivation of any of the HECAs of the NVP/2/1 architecture. It is noteworthy that, in each case, the reconfiguration affects only a single HECA. The decision has to be

⁷ An enumeration of all the possible combinations reveals 18 solutions.

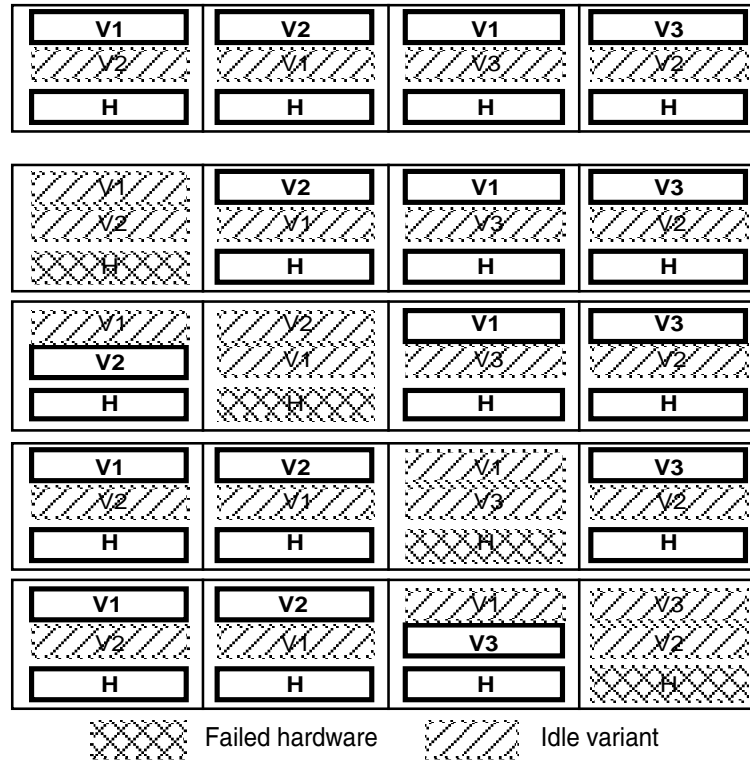


Figure 3.13 Various activations of the variants in the NVP/2/1 Architecture

modified to a vote among the remaining variants and thus the degraded architecture is the same as the NVP/1/1 architecture.

3.4.5.4 THE NVP/2/2 ARCHITECTURE: AN EXAMPLE OF ARCHITECTURE TOLERATING TWO SOLID SOFTWARE FAULTS

To illustrate the impact of the solid character of the software faults, on the design of architectures tolerating two faults, we consider the case of the NVP method. The definition of such an architecture requires the provision of four disjoint HECAs and SECAs, hence the NVP/2/2 architecture.

Although, this architecture appears as a direct extension of the NVP/1/1 architecture by addition of one HECA and one associated SECA, major differences exist in the processing of the errors. The fault tolerance decision is now based on the identification, among the results provided by the four variants, of a single set of agreeing results made up of two or more results. Furthermore, after the first discrepancy among the results, the hardware component (and its associated variant) designated as failed is made passive without any attempt to diagnose the fault as a hardware or software fault. The decision is then modified as a vote among the remaining versions and the degraded architecture obtained is close to the NVP/1/1 architecture. However, the same discarding action as the one described above would be carried out upon manifestation of a subsequent fault.

Besides the tolerance to two consecutive independent software faults, this architecture allows one a) to tolerate two simultaneous independent faults, b) to detect related fault among two variants and c) to detect simultaneous independent faults in three or four variants.

The Fault-Tolerant Processor-Attached Processor (FTP-AP) architecture proposed in [Lal88] may be seen as an implementation of this hardware-and-software fault-tolerant architecture. In the paper, a quad configuration of the core FTP architecture is used as a support for the management of the execution of 4 diversified applications software that are run on 4 distinct application processors.

3.4.6 Summary of the Fault Tolerance Properties of the Architectures

Figure 3.14 summarizes the main fault tolerance properties of the architectures introduced in the preceding paragraphs. Besides the generic notation identifying the software fault tolerance method and the number of hardware faults and independent software faults tolerated, the table presents for each architecture, the number of hardware components and the number of variants required. When applicable, some properties in addition to nominal fault tolerance are listed as well. Finally, the rightmost field of the table gives the fault tolerance properties of the architecture that is obtained after tolerance of a hardware fault, that is, when a HECA has been made passive.

Architecture	# of hardware components	# of variants	Properties in addition to nominal fault tolerance		Fault tolerance properties after a HECA has been made passive	
			Hardware faults	Software faults	Hardware	Software
RB/1/1	2	2	Low error latency	—	Detection provided by local diagnosis	Tolerance of one independent fault
NSCP/1/1	4	4	Tolerance of 2 faults in hardware components of the same SECA Detection of 3 or 4 faults in hardware components	Tolerance of 2 independent faults in the same SECA Detection of 2 related faults in disjoint SECAs Detection of 2, 3 or 4 independent faults	Detection	Detection of independent faults
NSCP/1/1/m	4	3	Tolerance of 2 faults in hardware components of the same SECA	—	Detection	Detection of independent faults
NVP1/1	3	3	Detection of 2 or 3 faults	Detection of 2 or 3 independent faults	Detection	Detection of independent faults
RB/2/1	3	2	Low error latency	—	Identical to RB/1/1	
NSCP/2/1	6	3	Detection of 3 to 6 faults in hardware components	Detection of 2 or 3 independent faults	Identical to NSCP/1/1	
NVP/2/1	4	3	Detection of 3 or 4 faults in hardware components Tolerance of combinations of single fault in hardware component and independent fault in non-duplicated variant	Detection of 2 or 3 independent faults	Identical to NVP/1/1	
NVP/2/2	4	4	Detection of 3 or 4 faults in hardware components	Detection of 2 related faults Tolerance of 2 independent faults Detection of 3 or 4 independent faults	Identical to NVP/1/1	

Figure 3.14 Synthesis of the properties of the hardware-and-software fault-tolerant architectures

3.4.7 Dependability Analysis of Hardware-and-Software Fault-Tolerant Architectures

The aim of this section is to show how, based on the results of Section 3.1, a dependability analysis of hardware-and-software fault-tolerant architectures can be conducted when adopting a Markov approach. Other related studies of special interest include: a) the application of a similar modeling approach to actual data derived from the experimental implementation of a real-world autopilot [Dug94], and b) the comprehensive performance and dependability (performability) analysis of fault-tolerant software carried out in [Tai93].

In what follows, three architectures are considered that enable to tolerate a single hardware or software fault, i.e., RB/1/1, NSCP/1/1 and NVP/1/1.

3.4.7.1 MODELING METHOD AND ASSUMPTIONS

To model the behavior of the architectures the following is assumed:

- only one type of fault(s) can be activated at each execution and produce error(s), either hardware or software,
- after detection and recovery of an error, the variant is not discarded and at the next step it is supplied with the new input data, i.e., faults are considered to be soft faults,
- constant failure rates are considered for the hardware components and the various sources of failure of the software-fault tolerant architectures.

Although the use of constant failure rates is quite common for hardware, the use of constant failure rates for the software-fault tolerant architectures is justified by the results presented in [Arl88], which can be summarized as follows:

- since a failed variant is not discarded but merely restarted at next execution, the state graphs of the failure behavior of the considered fault-tolerant software components are strongly connected.
- as a consequence, when constant execution rates are considered for the various activities (activation, variant(s) execution and decision(s)) of the fault-tolerant software, it is possible to define equivalent constant failure rates which are obtained as the product of the mean activation rate of the considered fault-tolerant software by the probabilities of failure.

3.4.7.2 MODELS

Architecture behavior can be modeled by the state diagrams shown on Figure 3.15-a,-b and -c for the RB/1/1, NSCP/1/1 and the NVP/1/1 architectures, respectively. For the RB/1/1 architecture, hardware and software fault tolerance mechanisms are independent: a hardware failure does not alter the software fault tolerance capabilities and vice versa. It is assumed that a near-perfect detection coverage can be achieved for hardware faults since both HECAs run simultaneously the same variant, thus, the coverage considered here for the hardware-fault tolerance mechanisms corresponds to a localization coverage due to a) the diagnosis program and b) the capacity of the acceptance test to identify hardware failures.

In the case of the NSCP/1/1 architecture, hardware and software fault tolerance techniques are not independent since the HECAs and the SECAs match. After the failure of a hardware component, the corresponding HECA (and hence, SECA) is discarded; the remaining archi-

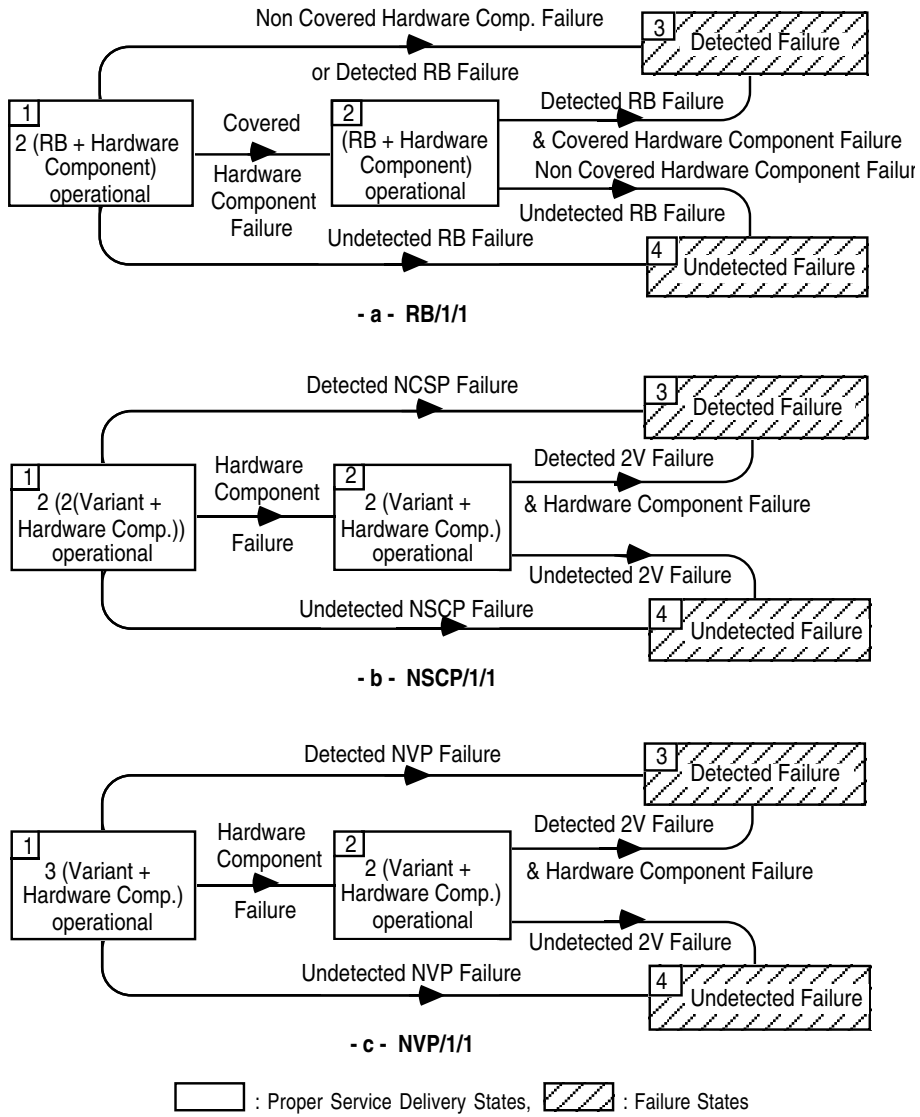


Figure 3.15 State diagrams of the architectures considered

ture is composed of a pair of hardware components and a two-version software architecture that form a hardware- and software-fault self-checking architecture.

In the case of the NVP/1/1 architecture, again, hardware and software fault tolerance techniques are not independent: after a hardware unit has been made passive, the remaining architecture is analogous to the degraded architecture of the NSCP/1/1 architecture.

For both NSCP/1/1 and NVP/1/1 architectures, the hardware faults are tolerated at the software layer through the decision algorithm (comparison or vote); accordingly, the associated coverage will be accounted for at the software level only and thus incorporated in the probability of activation of a fault in the decider.

In the degraded architectures obtained from NSCP/1/1 and NVP/1/1 after failure of a hardware component, the software is not any more fault tolerant and thus the failure rates of the variants are of significant importance in the failure rate of the degraded configuration of the application software.

The transition rate from state i to state j is denoted T_{ij} . The expressions of the inter-state transitions of Figure 3.15 are given by Figure 3.16.

	RB / 1 / 1	NSCP / 1 / 1	NVP / 1 / 1
T₁₂	$2 c \lambda_{H, RB}$	$4 \lambda_{H, NSCP}$	$3 \lambda_{H, NVP}$
T₁₃	$2 \bar{c} \lambda_{H, RB} + \lambda_{S, D, RB}$	$\lambda_{S, D, NSCP}$	$\lambda_{S, D, NVP}$
T₁₄	$\lambda_{S, U, RB}$	$\lambda_{S, U, NSCP}$	$\lambda_{S, U, NVP}$
T₂₃	$c \lambda_{H, RB} + \lambda_{S, D, RB}$	$2 \lambda_{H, NSCP} + \lambda_{S, D, 2V}$	$2 \lambda_{H, NVP} + \lambda_{S, D, 2V}$
T₂₄	$\bar{c} \lambda_{H, RB} + \lambda_{S, U, RB}$	$\lambda_{S, U, 2V}$	$\lambda_{S, U, 2V}$

Figure 3.16 Transitions and associated failure rates

The notation is as follows:

- c is the hardware coverage factor of the RB/1/1 architecture, $\bar{c} = 1 - c$,
- $\lambda_{H, X}$ denotes the failure rate for a hardware component of the architecture X/1/1, $X \in \{RB, NSCP, NVP\}$,
- $\lambda_{S, D, X}$ and $\lambda_{S, U, X}$ denote respectively the detected and undetected failure rates for the fault tolerant software X, $X \in \{RB, NSCP, NVP\}$; if γ denote the application software activation rate, then these failure rates can be expressed as functions of the failure probabilities given in Figure 3.8 (Section 3.3.1.5) as:

$$\lambda_{S, D, X} = [q_{S, D, X}] \gamma \quad \text{and} \quad \lambda_{S, U, X} = [q_{S, U, X}] \gamma \quad (3.2)$$

- $\lambda_{S, D, 2V}$ and $\lambda_{S, U, 2V}$ denotes respectively the application software detected and undetected failure rates of the NSCP/1/1 and NVP/1/1 architectures, after an HECA has been made passive; these rates are defined as:

$$\lambda_{S, D, 2V} = [q_{S, D, 2V}] \gamma \quad \text{and} \quad \lambda_{S, U, 2V} = [q_{S, U, 2V}] \gamma \quad (3.3)$$

where the detected and undetected failure probabilities of the degraded 2V configuration are defined by:

$$q_{S, D, 2V} = 2q_{I, 2V} \left(1 - \frac{q_{I, 2V}}{2}\right) + q_{ID, 2V} \quad \text{and} \quad q_{S, U, 2V} = q_{RVD, 2V} \quad (3.4)$$

3.4.7.3 MODEL PROCESSING

Processing of the models of Figure 3.15, when accounting for the different transition rates of Figure 3.16, enables one to derive the expressions for the time-dependent probabilities of detected failure and undetected failure of the considered hardware-and-software fault-tolerant architectures: $Q_{D,X}(t)$ and $Q_{U,X}(t)$, respectively, $X \in \{RB, NSCP, NVP\}$. In practice, two types of probabilities are of interest:

- the reliability, $R_X(t) = Q_{D,X}(t) + Q_{U,X}(t)$,
- the probability of undetected failure, $Q_{U,X}(t)$.

For missions of short duration with respect to the mean times to failure, these expressions can be simplified to the following approximate expressions:

$$R_{RB}(t) \approx 1 - (2\bar{c}\lambda_{H,RB} + \lambda_{S,RB})t \quad Q_{U,RB}(t) \approx \lambda_{S,U,RB}t \quad (3.5)$$

$$R_{NSCP}(t) \approx 1 - \lambda_{S,NSCP}t \quad Q_{U,NSCP}(t) \approx \lambda_{S,U,NSCP}t \quad (3.6)$$

$$R_{NVP}(t) \approx 1 - \lambda_{S,NVP}t \quad Q_{U,NVP}(t) \approx \lambda_{S,U,NVP}t \quad (3.7)$$

These expressions show that — as could be expected — the reliability of the RB/1/1 architecture is strongly dependent on the coverage of the fault diagnosis in the hardware components. Furthermore, it is likely that the hardware component failure rate be greater for the RB/1/1 architecture than for the other ones, due to the extra memory needed to store the second variant; also further hardware and/or software resources would be needed to perform the comparison among the results provided by each hardware processor ensuring thus a near-perfect detection coverage and storage would be needed for the acceptance test and the diagnosis program.

The expressions also reveal an identical influence of the failure rate of the application software for the three architectures. However, this has to be tempered by the differences between the associated probabilities identified in Section 3.3.1.5.

3.5 CONCLUSIONS

In this chapter, we have a) presented in a unified way the three major approaches for tolerating software faults by design diversity, and b) defined and analyzed architectures aimed at tolerating both hardware and software faults.

In addition to recovery block and N -version programming that form the basic architectures to implement software-fault tolerance, we have introduced and analyzed the N self-checking programming scheme that corresponds to the architecture actually implemented in real-life safety-critical applications.

The careful analysis of these architectures enabled to identify their major dependability features in particular concerning their ability to cope with independent and related software fault combinations in the variants and the deciders.

The extension of the notion of error confinement areas as a structuring principle to hardware- and software-fault tolerant systems made it possible a) to carry out a systematic study of the fault tolerance features provided by the considered architectures, and b) to derive comprehensive analytical models for the evaluation of system-level dependability measures.

Finally, it is worth noting that emergence of hardware-fault tolerant commercial systems

will make more sensitive the influence of design faults on the service delivered by computing systems to their users. As a foreseeable consequence, software fault tolerance is likely to spread out its currently privileged domain, i.e., safety-related systems. The approaches and results presented in this chapter are thus likely to be accordingly of a wider field of application.

ACKNOWLEDGEMENTS

This work has been partially supported by Aérospatiale and by Matra Marconi Space in the framework of the HERMES project (European Space Shuttle) and by the CEC in the framework of the ESPRIT Project "Predictably Dependable Computing Systems".

Christian Béounes passed away on April 23, 1993, after several months of painful battle against illness. Christian lives on in our memory, both for his professional and for his human qualities.

REFERENCES

- [Amm87] P. E. Ammann and J. C. Knight. Data diversity: an approach to software fault tolerance. In *Proc. 17th International Symposium on Fault-Tolerant Computing (FTCS-17)*, pages 122–126, Pittsburgh, PA, July 1987. IEEE Computer Society Press.
- [And79] D. M. Andrews. Using executable assertions for testing fault tolerance. In *Proc. 9th International Symposium on Fault-Tolerant Computing (FTCS-9)*, pages 102–105, Madison, WI, June 1979. IEEE Computer Society Press.
- [And81] T. Anderson and P. A. Lee. *Fault Tolerance — Principles and Practice*. Prentice Hall, 1981. (See also: P.A. Lee, T. Anderson, *Fault Tolerance — Principles and Practice, Dependable Computing and Fault-Tolerant Systems*, volume 3, Springer-Verlag, Vienna, Austria, 1990).
- [And85] T. Anderson, P. A. Barrett, D. N. Halliwell, and M. Moulding. Software fault tolerance: an evaluation. *IEEE Transactions on Software Engineering*, SE-11(12):1502–1510, 1985.
- [Arl88] J. Arlat, K. Kanoun, and J.-C. Laprie. Dependability evaluation of software fault-tolerance. In *Proc. 18th International Symposium on Fault-Tolerant Computing (FTCS-18)*, pages 142–147, Tokyo, Japan, June 1988. IEEE Computer Society Press. (See also: Dependability Modeling and Evaluation of Software Fault-Tolerant Systems. *IEEE Transactions on Computers*, 39(4):504–513, 1990).
- [Arn73] T. F. Arnold. The concept of coverage and its effect on the reliability model of repairable systems. *IEEE Transactions on Computers*, C-22(3):251–254, 1973.
- [Avi79] A. Avižienis. Towards a discipline of reliable computing. In *Proc. EURO IFIP'79*, pages 701–705, London, UK, September 1979.
- [Avi84] A. Avižienis and J. P. J. Kelly. Fault tolerance by design diversity: concepts and experiments. *IEEE Computer*, 17(8):67–80, 1984.
- [Avi85] A. Avižienis. The N-version approach to fault-tolerant systems. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985.
- [Avi86] A. Avižienis and J.-C. Laprie. Dependable computing: from concept to design diversity. *Proceedings of the IEEE*, 74(5):629–638, 1986.
- [Bis86] P. G. Bishop, D. G. Esp, M. Barnes, P. Humphreys, G. Dahl, and J. Lahti, PODS—a project on diverse software. *IEEE Transactions on Software Engineering*, 12(9):929–940, 1986.
- [Boe81] B. W. Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ, 1981.
- [Bon93] A. Bondavalli, F. Di Giandomenico, and J. Xu. A cost-effective and flexible scheme for software fault tolerance. *International Journal of Computer Systems Science & Engineering*, 8:234–244, 1993.
- [Bou69] W. G. Bouricius, W. C. Carter, and P. R. Schneider. Reliability modeling techniques for self-repairing computer systems. In *Proc. 24th. National Conference*, pages 295–309, ACM, 1969.

- [Bri93] D. Brière and P. Traverse. AIRBUS A320/A330/A340 electrical flight controls —a family of fault-tolerant systems. In *Proc. 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 616–623, Toulouse, France, June 1993. IEEE Computer Society Press.
- [Cag85] A. K. Caglayan and D. E. Eckhardt. System approach to software fault tolerance. In *Proc. 5th Computers in Aerospace Conference*, pages 361–369, Long Beach, CA, IEEE/AIAA, October 1985.
- [Cau81] J.T. Caulfield. Application of redundant processing to space shuttle. In *Proc. 8th IFAC Triennial World Congress*, pages 2461–2466, Kyoto, Japan, 1981.
- [Che78] L. Chen and A. Avižienis. N-version programming: a fault tolerance approach to reliability of software operation. In *Proc. 8th International Symposium on Fault-Tolerant Computing (FTCS-8)*, pages 3–9, Toulouse, France, June 1978. IEEE Computer Society Press.
- [Chr94] J. Christmansson, Z. Kalbarczyk, and J. Torin. Dependable flight control system using data diversity with error recovery. *International Journal of Computer Systems Science & Engineering*, 9:98–106, April 1994.
- [Dug94] J. B. Dugan and M. R. Lyu. System-level reliability and sensitivity analyses for three fault-tolerant system architectures. In *4th International Working Conference on Dependable Computing for Critical Applications*, pages 295–307, San Diego, CA, January 1994.
- [Eck91] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk, and J. P. J. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions on Software Engineering*, 17(7):692–702, 1991.
- [Elm72] W. R. Elmendorf. Fault-tolerant programming. In *Proc. 2nd International Symposium on Fault-Tolerant Computing (FTCS-2)*, pages 79–83, Newton, MA, June 1972. IEEE Computer Society Press.

- [Fis75] M. A. Fishler, P. Firshein, and D. L. Drew. Distinct software: an approach to reliable computing. In *Proc. 2nd USA-Japan Computer Conference*, pages 573–579, Tokyo, Japan, August 1975. IEEE Computer Society Press.
- [Gar81] J. R. Garman. The “bug” heard around the world. *ACM Sigsoft Software Engineering Notes*, 6(5):3–10, 1981.
- [Gme79] L. Gmeiner and U. Voges. Software diversity in reactor protection systems: an experiment. In *Proc. International IFAC Workshop on Safety on Computer Control systems (SAFE-COMP’79)*, pages 89–93, Stuttgart, Germany, 1979.
- [Gra86] J. Gray. Why do computers stop and what can be done about it? In *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, Los Angeles, CA, IEEE, 1986.
- [Gra90] J. Gray. A census of Tandem system availability between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409–432, 1990.
- [Hag86] G. Hagelin. ERICSSON safety system for railway control. Technical Report ENR/TB 6078, ERICSSON, October 1986. Also in *Application of Design Diversity in Computerized Control Systems*. In *Proceedings of the IFIP WG 10.4 Workshop on Design Diversity in Action*, Baden, Austria, June 1986, U. Voges, editor, volume 2 of the Series on Dependable Computing and Fault Tolerance, pages 11–21, Springer-Verlag, Vienna, Austria, 1988.
- [Kel88] J. P. J. Kelly, D. E. Eckhardt, M. A. Vouk, D. F. Allister, and A. K. Caglayan. A large scale second generation experiment in multi-version software: description and early results. In *Proc. 18th International Symposium on Fault-Tolerant Computing (FTCS-18)*, pages 9–14, Tokyo, Japan, June 1988. IEEE Computer Society Press.
- [Kim84] K. H. Kim. Distributed execution of recovery blocks: an approach to uniform treatment of hardware and software faults. In *Proc. 4th International Conference on Distributed Computing Systems*, pages 526–532, May 1984. IEEE Computer Society Press.
- [Kim89] K. H. Kim and H. O. Welch. Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Transactions on Computers*, C-38(5):626–636, 1989.
- [Kni86] J. C. Knight and N. G. Leveson. An empirical study of failure probabilities in multi-version software. In *Proc. 16th International Symposium on Fault-Tolerant Computing (FTCS-16)*, pages 165–170, Vienna, Austria, July 1986. IEEE Computer Society Press.
- [Lal88] J. H. Lala and L. S. Alger. Hardware and software fault tolerance: a unified architectural approach. In *Proc. 18th International Symposium on Fault-Tolerant Computing (FTCS-18)*, pages 240–245, Tokyo, Japan, June 1988. IEEE Computer Society Press.
- [Lap87] J.-C. Laprie, J. Arlat, C. B’éounes, K. Kanoun, and C. Hourtolle. Hardware and software fault-tolerance: definition and analysis of architectural solutions. In *Proc. 17th International Symposium on Fault-Tolerant Computing (FTCS-17)*, pages 116–121, Pittsburgh, PA, 1987. IEEE Computer Society Press.
- [Lap90] J.-C. Laprie, J. Arlat, C. B’éounes, and K. Kanoun. Definition and analysis of hardware-and-software fault-tolerant architectures. *IEEE Computer*, 23:39–51, July 1990.
- [Lap94] J.-C. Laprie, J. Arlat, C. B’éounes, and K. Kanoun. Fault tolerant computing. In *Encyclopedia of Software Engineering, Vol. 1 A-N*, J. Marciniak, editor, pages 482–503, J. Wiley & Sons, 1994.
- [Lor89] P. R. Lorczak, A. K. Caglayan, and D. E. Eckhardt. A theoretical investigation of generalized voters for redundant systems. In *Proc. 19th International Symposium on Fault Tolerant Computing (FTCS-19)*, pages 444–451, Chicago, IL, June 1989. IEEE Computer Society Press.
- [Mah84] A. Mahmood, D. M. Andrews, and E. J. McCluskey. Executable assertions and flight software. In *Proc. 6th Digital Avionics Systems Conference*, pages 346–351, Baltimore, Maryland, AIAA/IEEE, December 1984.
- [Mar82] D. J. Martin. Dissimilar software in high integrity applications in flight controls. In *Proc. AGARD CP-330*, pages 36.1–36.13, September 1982.
- [Neu86] P. G. Neumann. On hierarchical design of computer systems for critical applications. *IEEE Transactions on Software Engineering*, SE-12(9):905–920, 1986.
- [Ram84] C. V. Ramamoorthy, A. Prakash, W.-T. Tsai, and Y. Usuda. Software engineering: problems and perspectives. *IEEE Computer*, 17(10):191–209, 1984.

- [Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, 1975.
- [Ran84] B. Randell. Fault tolerance and system structuring. In *Proc. 4th Jerusalem Conference on Information Technology (JCIT-4)*, pages 182–191, Jerusalem, Israel, May 1984.
- [Ran87] B. Randell. Design fault tolerance. In *The Evolution of Fault-Tolerant Computing* A. Avižienis, H. Kopetz, and J.-C. Laprie, editors, volume 1 of *Dependable Computing and Fault-Tolerant Systems*, pages 251–270, Vienna, Austria: Springer-Verlag, 1987. (Also *Proc. IFIP Symposium on The Evolution of Fault-Tolerant Computing*, Baden, Austria, June 30, 1986).
- [Rem82] L. Remus. Methodology for software development of a digital integrated protection system. January 1982. (Presented at the EWICS TC-7, Brussels).
- [Rou86] J. C. Rouquet and P. Traverse. Safe and reliable computing on board Airbus and ATR aircraft. In *Proc. 5th IFAC Workshop Safety of Computer Control Systems (SAFECOMP'86)*, W. J. Quirk, editor, pages 93–97, Sarlat, France, October 1986. Pergamon Press.
- [Sco87] R. K. Scott, J. W. Gault, and D. F. McAllister. Fault-tolerant software reliability modeling. *IEEE Transactions on Software Engineering*, SE-13(5):582–592, 1987.
- [She78] C. T. Sheridan. Space shuttle software. *Datamation*, pages 128–140, 1978.
- [Sie82] D. P. Siewiorek and D. Johnson. A design methodology for high reliability systems: the Intel 432. In D. P. Siewiorek and R. S. Swarz, editors, *The Theory and Practice of Reliable System Design*, pages 621–636, 1982. Digital Press, Bedford, MA.
- [Spr80] R. E. Spradlin. Boeing 757 and 767 flight management system. In *Proc. RTCA Tech. Symposium*, pages 107–118, Washington, DC, November 1980.
- [Ste78] B. J. Sterner. Computerized interlocking system — a multidimensional structure in pursuit of safety. *IMechE Railway Engineer International*, pages 29–30, 1978.
- [Sul90] G. F. Sullivan and G. M. Masson. Using certification trails to achieve software fault tolerance. In *Proc. 20th International Symposium on Fault-Tolerant Computing (FTCS-20)*, pages 423–431, Newcastle upon Tyne, UK, June 1990. IEEE Computer Society Press.
- [Swa76] E. B. Swanson. The dimension of maintenance. In *Proc. 2nd International Conference on Software Engineering*, pages 492–497, Los Alamitos, CA, 1976. IEEE Computer Society Press.
- [Tai93] A. T. Tai, J. F. Meyer, and A. Avižienis. Performability enhancement of fault-tolerant software. *IEEE Transactions on Reliability*, 42(2):227–237, 1993.
- [Toy78] W. N. Toy. Fault-tolerant design of local ESS processors. *Proceedings of the IEEE*, 66(10):1126–1145, 1978.
- [Tra88] P. Traverse. AIRBUS and ATR system architecture and specification. In *Software Diversity in Computerized Control Systems*, U. Voges, editor, volume 2 of *Dependable Computing and Fault-Tolerant Systems*, pages 95–104, Vienna, Austria: Springer-Verlag, 1988.
- [Tso86] K. S. Tso and A. Avižienis. Error recovery in multi-version software. In *Proc. 5th IFAC Workshop Safety of Computer Control Systems (SAFECOMP'86)*, W. J. Quirk, editor, pages 35–41, Sarlat, France, October 1986. Pergamon Press.
- [Tso87] K. S. Tso and A. Avižienis. Community error recovery in N-version software: a design study with experimentation. In *Proc. 17th International Sym. on Fault-Tolerant Computing (FTCS-17)*, pages 127–133, Pittsburgh, PA, July 1987. IEEE Computer Society Press.
- [Vog88] U. Voges. Software diversity in computerized control systems. In *Dependable Computing and Fault-Tolerant Systems*, A. Avižienis, H. Kopetz, and J.-C. Laprie, editors, volume 2, Vienna, Austria: Springer-Verlag, 1988. (Also *Proc. IFIP Workshop on Design Diversity in Action, Baden, Austria*, June 28, 1986).
- [Xu91] J. Xu. The $t/(n - 1)$ -diagnosability and its applications to fault tolerance. In *Proc. 21st International Symposium on Fault Tolerant Computing (FTCS-21)*, pages 496–503, Montré'al, Quebec, Canada, June 1991. IEEE Computer Society Press.
- [Yau75] S. S. Yau and R. C. Cheung. Design of self-checking software. In *Proc. International Conference on Reliable Software*, pages 450–457, Los Angeles, CA, April 1975. IEEE Computer Society Press.
- [You84] L. J. Yount. Architectural solutions to safety problems in digital flight-critical systems for commercial transports. In *Proc. 6th Digital Avionics Conference*, pages 28–35, Baltimore, MD, December 1984.

- [Zel79] M. Zelkowitz, A. C. Shaw, and J. D. Gannon, *Principles of Software Engineering and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1979.