Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures

Jean-Claude Laprie, Jean Arlat, Christian Béounes,

and Karama Kanoun

LAAS-CNRS

Both experimental and real-life safety-related systems have begun to use design diversity to tolerate software faults.[1] Such systems focus strongly on design faults, where the term "design" encompasses everything from system requirements to realization during both initial production and future modifications. Design faults are a source of common-mode failures, which defeat fault-tolerance strategies based on strict replication (that cope with physical faults) and generally have catastrophic consequences.

Precomputer safety-related systems minimized common-mode failures through diversified design, that is, two or more systems delivering the same service through separate designs and realizations. A typical example is a hardwired electronic channel backed by an electromechanic or electropneumatic channel. In addition, system architecture was based on the federation of equipment, where each piece of equipment implemented one or more subfunctions of the system rather than the entire system. Such partitioning confined equipment failures to subfunctions, allowing the system's global function to continue, although possibly in a degraded mode.

Systems in which one piece of hardware supports multiple software are subject to software failures and require architectures that tolerate both hardware and software faults.

Computer-based safety-related systems generally retain the federation approach. Each subfunction is implemented by a "complete" computer comprising hardware and executive and application software. Examples of this approach include airplane flight-control systems (such as in the Boeing 757/767 airliner) and nuclear-plant monitors (such as Merlin-Gérin's Système de Protection Intégré Numérique).

To confine computer failures, a system must automatically check execution results for the errors that could lead to failure. There are two main approaches to detecting errors caused by design faults:

(1) Acceptance tests of the results via executable assertions. These assertions are generalized, formalized versions of likelihood checks used in process control.
(2) Diversified design, so that the results of two software variants can be compared (as in the Airbus A-300 and A-310 airliners and the Swedish railways' interlocking system).[1]

The federation approach generally requires far more processing elements than are needed for computing power alone; for instance, the Boeing 757/767 flight-control system comprises 80 distinct functional microprocessors, 300 when we account for redundancy.

We could use computers better in such systems if the same hardware supported software for several subfunctions. Such an approach, called integration, is subject to software failures, which are due to design faults only. Thus, integration requires software-fault tolerance. Moreover, some safety-related systems (such as those in the

Table 1. Main characteristics of the software-fault-tolerance strategies.

Method	Error-Processing Technique	Judgment on Result Acceptability	Variant-Execution Scheme	Consistency of Input Data	Suspension of Service Delivery During Error Processing	No. Variants to Tolerate f Sequential Faults
Recovery Blocks (RB)	Error detection by acceptance tests 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					
	Detection by acceptance tests	Absolute, with respect to specification	Parallel	Explicit, by dedicated mechanisms	Yes, duration necessary for result switching	$f+1$
	Detection by comparison	Relative, on variant results	Parallel	Explicit, by dedicated mechanisms	Yes, duration necessary for result switching	$2(f+1)$
N-Version Programming (NVP)	Vote	Relative, on variant results	Parallel	Explicit, by dedicated mechanisms	No	$f+2$

NASA Space Shuttle and the Airbus A-320 airliner) are moving toward limiting or eliminating manual or noncomputer backup systems. This is an additional incentive for software-fault tolerance, since safe system behavior becomes entirely dependent on reliable software behavior.

This article elaborates on previous work to present a structured definition of hardware- and software-fault-tolerant architectures.[2] We have tried to be as general as possible, dealing with specific classes of faults or techniques only when necessary. (More specific definitions extending the recovery block approach[3] and N-version programming[4] have appeared elsewhere.) After discussing software-fault-tolerance methods, we present a set of hardware- and software-fault-tolerant architectures and analyze and evaluate three of them. A sidebar addresses the cost issues related to software-fault tolerance.

Software-fault-tolerance methods

In a diversified design, the different systems produced from a common service specification are called *variants*. A diversified design has at least two variants plus a *decider*, which monitors the results of variant execution, given consistent initial conditions and inputs. The common specification must explicitly address the decision points, that is, it must state when to make decisions and what data to base them on (the data processed by the decider).

The best-documented techniques for tolerating software design faults are the *recovery block* (RB) approach[5] and *N-version programming* (NVP).[6] In the first approach, the variants are called alternates and the decider is an acceptance test, which is applied sequentially to the alternates' results. If the results of the primary alternate do not satisfy the acceptance test, the secondary alternate executes. In the second approach, the variants are called versions, and the decider is a vote based on all versions' results.

We use the term "variant" rather than "alternate" or "version" because "alternate" reflects sequential execution, which is a feature specific to the recovery block approach, and "version" has another meaning: successive versions of a system resulting from fault removal or functionality evolution. During the life of a diversely designed system, several versions of the variants will be generated.

The hardware-fault-tolerant architectures equivalent to RB and NVP are standby sparing and N-modular redundancy, respectively. A third approach to hardware-fault tolerance, active dynamic redundancy, is very popular (especially when based on self-checking components, such as in the AT&T Electronic Switching System and the Stratus system), but it has not been described in the literature as a generic technique for software-fault tolerance. However, self-checking programming has long been defined;[7] a self-checking program results from adding redundancy to a program so that it can check its own dynamic behavior during execution. A self-checking software component consists of either a variant and an acceptance test or two variants and a comparison algorithm.

Fault tolerance is provided by the parallel execution of at least two self-checking components. At each execution of such a system, one component "acts" (that is, it delivers service or results to the controlled or monitored application), while the other components remain "hot" spares. When the acting component fails, a spare begins delivering service. If a spare fails, the acting component continues delivering service. Error processing is thus performed through error detection and possible switching of results. We call this approach *N self-checking programming* (NSCP).

It could be argued that NSCP is just a parallel recovery block scheme, but the latter's backward recovery strategy prevents it from being reduced to the association of alternates together with an acceptance test. In NSCP, when a self-checking

Table 2. Overheads for tolerance of one software fault.

Method	Structural Overhead		Operational Time Overhead		
	Diversified Software Layer	Mechanisms	Systematic		When Errors Occur
			Decider	Variants Execution	
Recovery Blocks	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					
Error detection by acceptance tests	One variant and two acceptance tests	Result switching	Acceptance test execution	Input data consistency and variants execution synchronization	Possible result switching
Error detection by comparison	Three variants	Comparators and result switching	Comparison execution	Input data consistency and variants execution synchronization	Possible result switching
N-Version Programming	Two variants	Voters	Vote execution	Input data consistency and variants execution synchronization	Usually negligible

software component is based on the association of two variants, only one variant fulfills the expected functions, while the other acts as an extended acceptance test. Each self-checking component in NSCP is responsible for determining whether a delivered result is acceptable, whereas the judgment of acceptability in NVP is cooperative. Also, each acceptance test associated with a variant, or each comparison algorithm associated with a pair of variants, can be the same or can be specifically derived from a common specification for each variant or variant pair. As in N-version programming, the components' parallel execution necessitates a mechanism to ensure input consistency.

Our aim in this article is to classify the various approaches to software-fault tolerance, not to introduce a new approach. In fact, most of the real-life systems mentioned in the introduction do not implement either a recovery block approach or N-version programming, but rather are based on self-checking software. For instance, the Airbus A-300 and A-310 flight-control systems and the Swedish railways' interlocking system are based on the parallel execution of two variants that stop operation when a comparison of their results reveals an error. The Airbus A-320 flight-control system is based on two self-checking components, each based in turn on the parallel execution of two variants whose results are compared. Tolerance of a single fault in this system requires four variants. (The two self-checking compo-

nents in this last scheme do not deliver exactly the same service. Critical functions are preserved when the system switches from the acting component to the spare, but noncritical functions are performed in a degraded mode.)

Table 1 summarizes the main characteristics of the three strategies. In selecting a strategy for a given application, pay particular attention to the method for judging result acceptability and whether service delivery is suspended when an error occurs. Table 2 summarizes the main sources of structural and operation-time overhead for software-fault tolerance. The table does not mention overhead imposed by tests local to each variant, such as input range checking and grossly wrong results, since such tests are common to all approaches (and are — or should be — present in nonfault-tolerant software systems, as well).

Fault classes. We classify faults according to their independence and their persistence.

Independence. Faults are either related or independent.[6] Related faults result from a specification fault common to all variants or from dependencies in the separate designs and implementations. Independent faults are simply those that are not related. Related faults manifest themselves as similar errors and lead to common-mode failures, whereas independent faults usually cause distinct errors and separate failures. Figure 1 illustrates these definitions.

Persistence. Faults are classified as solid or soft based on their persistence. Such a distinction is usual in hardware, where a fault's solidity or softness is important to fault tolerance. A component affected by a solid fault must be made passive after the fault is detected, whereas a component affected by a soft fault can be used after error recovery. In other words, a solid fault necessitates error processing and fault treatment, while a soft fault requires only error processing. A permanent fault is a typical solid fault, and a tempo-
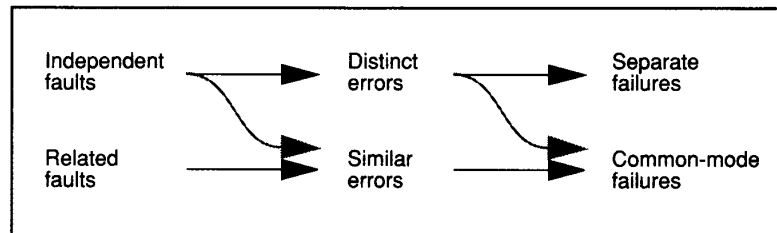

Figure 1. Classes of faults, errors, and failures.

The cost of software-fault tolerance

Fault tolerance introduces additional costs; we estimate those costs here. Since design diversity affects costs differently according to the life-cycle phases, we start with cost distribution among the various life-cycle activities for classical, non-fault-tolerant, software. Our simplified life-cycle model[1] (see the first table) groups all activities relating to verification and validation (V&V) separately.

Three maintenance categories cover the software's entire operational life.[1] Corrective maintenance concerns fault removal and involves design, implementation, and V&V. Adaptive maintenance adjusts software to environmental changes and also involves specification activity. Perfective maintenance improves the software's function; thus, it actually concerns software evolution, and so involves all development activities, starting with modified requirements.

The cost breakdowns for the life-cycle and maintenance[1] do not address a specific class of software. However, since we are concerned with critical applications, we must incorporate some multiplicative factors that depend on the particular activity.[2] The last two columns, which are derived from the data in the other columns, give the life-cycle cost distribution for development only and for development and maintenance.

Software cost elements for non-fault-tolerant software.

Activity	Life-Cycle Cost Breakdown[1]	Multipliers for Critical Applications[2]	Cost Distribution	
			Development	Development and Maintenance
Development				
Requirements	3%	1.3	8%	6%
Specification	3%	1.3	8%	7%
Design	5%	1.3	13%	14%
Implementation	7%	1.3	19%	19%
Verification				
and Validation	15%	1.8	52%	54%
Maintenance*	67%			

* Of this, 20% is for corrective maintenance, 25% is for adaptive maintenance, and 55% is for perfective maintenance.[1]

From this table, it appears that maintenance does not significantly affect cost distribution over the other life-cycle activities (in fact, the discrepancy is likely to be lower than indicated). Accordingly, let's assume in the following example that the figures for development only are general and cover the entire life-cycle, since we are concerned only with relative costs.

To determine the cost of fault-tolerant software, we must introduce factors to account for the overheads associated with the decision points and the deciders and to account for the cost reduction in V&V caused by commonalties among variants. These commonalties include actual V&V activities, such as back-to-back testing, and V&V tools, such as test harnesses. We cannot accurately estimate such factors given the current state of the art. We can, however, give reasonable ranges of variations.

Consider the following factors:
• r is the multiplier associated with the decision points, with $1 < r < 1.2$.
• s is the multiplier associated with the decider, with $1 < s < 1.1$ for NVP and NSCP when error detection is performed through comparison, and $1 < s < 1.3$ for RB and NSCP when error detection is performed through acceptance tests. This difference reflects the differences in the deciders, that is, the fact that the deciders are specific when

---

rary fault (either transient or intermittent) is a typical soft fault.

Let's now consider software faults in operational programs. Once a program has been thoroughly debugged, problems are more likely to arise from subtle fault conditions (such as limit conditions, race conditions, or strange underlying hardware conditions) than from easily identifiable faults. Just a slight change in the execution context could keep fault conditions from occurring again, thus keeping the software from failing again. Since the likelihood of such an error occurring again is negligible, we can extend the notion of a soft fault to software.[8]

Another important consideration for error recovery is the notion of local and global variables for the components. Let's call the program between two decision points a diversity unit. Generally, error recovery requires that the diversity units be procedures (so their activation and behavior do not depend on any internal

state). In other words, all data needed by a diversity unit must be global data. The data's global nature can result from the nature of the application itself. One example is physical-process monitoring (such as nuclear-plant protection), where tasks begin based on sensor data and do not use data from previous processing. The data's global nature can also result from transforming local data into global data. This incurs overhead and could decrease diversity (since the decision-point specification must be more precise). A simplified example is a filtering function that constitutes a diversity unit. In this example, past samples should be part of the global data.

Although these classifications apply to all software-fault-tolerance methods, we can alter the general rules somewhat in specific, application-dependent cases. For example, there is an alternate solution for NSCP and NVP when the overhead cannot be afforded or when transforming local data into global data will decrease diver-

sity too much. This solution involves fault treatment, that is, it eliminates failed variants from further processing.

Let's summarize the preceding discussion by adopting the following definitions for soft and solid faults: A soft software fault has a negligible likelihood of recurrence and is recoverable, while a solid software fault is recurrent under normal operation or cannot be recovered.

Defining hardware- and software-fault-tolerant architectures

Our discussion of architectures that tolerate both hardware and software faults emphasizes the dependencies among the software- and hardware-fault-tolerance methods and the effects of solid and soft software faults on the architecture definition. We investigate two levels of fault-

Cost of fault-tolerant software versus non-fault-tolerant software.

Faults Tolerated	Fault-Tolerance Method	N	(C_{FT}/C_{NFT})min	(C_{FT}/C_{NFT})max	(C_{FT}/C_{NFT})av	(C_{FT}/NC_{NFT})av
1	Recovery blocks	2	1.33	2.17	1.75	.88
1	N self-checking programming					
	Acceptance test	2	1.33	2.17	1.75	.88
	Comparison	4	2.24	3.77	3.01	.75
1	N-version programming	3	1.78	2.71	2.25	.75
2	Recovery blocks	3	1.78	2.96	2.37	.79
2	N self-checking programming					
	Acceptance test	3	1.78	2.96	2.37	.79
	Comparison	6	3.71	5.54	4.63	.77
2	N-version programming	4	2.24	3.77	3.01	.75

they decide by acceptance test and generic when they decide by comparison or vote.

• u is the proportion of testing performed once for all variants (such as provision for test environments and harnesses), with $0.2 < u < 0.5$.

• v is the proportion of testing, performed for each variant, that takes advantage of the existence of several variants (such as back-to-back testing), with $0.3 < v < 0.6$.

• w is the cost-reduction factor for testing performed in common for several variants, with $0.2 < w < 0.8$.

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

$$C_{FT}/C_{NFT} = \rho_{Req} + rs\,\rho_{Spe} +$$
$$[Nr + (s-1)]\,(\rho_{Des} + \rho_{Imp}) +$$
$$r\{us + (1-u)\,N\,[vw + (1-v)]\}\,\rho_{V\&V}$$

where N is the number of variants, and ρ_{Req}, ρ_{Spe}, ρ_{Des}, ρ_{Imp}, and $\rho_{V\&V}$ are

the cost distribution percentages for requirements, specification, design, implementation, and V&V, respectively.

The second table 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 distinguish between RB and NSCP with error detection by acceptance test, since our abstract cost model is likely to mask their differences.

The second table's results let us quantify the qualitative statement that N-variant software is less costly than N times a non-fault-tolerant software. Also note that previously published figures[3] fall within the ranges displayed here; that an experiment at the University of Newcastle upon Tyne estimated RB's overhead for two variants at 60 percent[3]; and that the Project on Diverse Software estimated the cost of NVP for three variants at 2.26 times the cost of a one-variant program.[3]

References

1. C.V. Ramamoorthy et al., "Software Engineering: Problems and Perspectives," *Computer*, Vol. 17, No. 10, Oct. 1984, pp. 191-209.

2. B.W. Boehm, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, N.J., 1981.

3. U. Voges, ed., "Application of Design Diversity in Computerized Control Systems," *Proc. IFIP Workshop on Design Diversity in Action*, Springer-Verlag, Vienna, 1986.

tolerance: architectures tolerating a single fault and architectures tolerating two consecutive faults. (We can relate these requirements, respectively, to the classical Fail Op/Fail Safe and Fail Op/Fail Op/Fail Safe requirements used in the aerospace community for hardware-fault tolerance.)

Due to the article's scope, our discussion is highly abstract. We do not discuss such distinguishing features as the overhead imposed by intercomponent communication for synchronization, decision-making, data consistency, etc., or the differences in memory space for each architecture.

Implementing design diversity. Of the many issues involved in design diversity,[6] two related issues are especially important: the number of variants and the level at which fault tolerance is applied.

Number of variants. Aside from economic considerations (see the sidebar), the number of variants for a given software-

fault-tolerance method is directly related to the number of faults to be tolerated (see Table 2). The soft or solid nature of the software faults significantly affects the architecture only when it must tolerate more than one fault. Also note that an architecture tolerating a solid fault can also tolerate a (theoretically) infinite sequence of soft faults, provided there are no fault coincidences.

The relation between the likelihood of such faults and the number of variants is not simple. Whether increasing the number of variants increases or decreases the number of related faults depends on several factors, some of which affect the others adversely.[6,9] However, there is good reason to increase the number of variants in NVP: in a three-version scheme, two similar errors can outvote a good result, while they would be detected in a four-version scheme.

Level of fault-tolerance application. The level of application involves two questions:

How much should the system be decomposed into components to be diversified? and Which layers (application software, executive, hardware) must be diversified?

The answer to the first question involves a trade-off between two opposing considerations: smaller components allow a better mastering of the decision algorithms, but larger components aid diversity. In addition, the decision points are "nondiversity" points (and synchronization points for NSCP and NVP); as such, they must be limited. Decision points are necessary only for interactions with the environment (sensor data acquisition, delivering orders to actuators, operator interaction, etc.). However, performance considerations could prompt additional compromises.

Concerning the second question, the methods for tolerating design faults can apply to any layer of either the application or the executive software. They can also apply to the hardware layers.[1] With respect
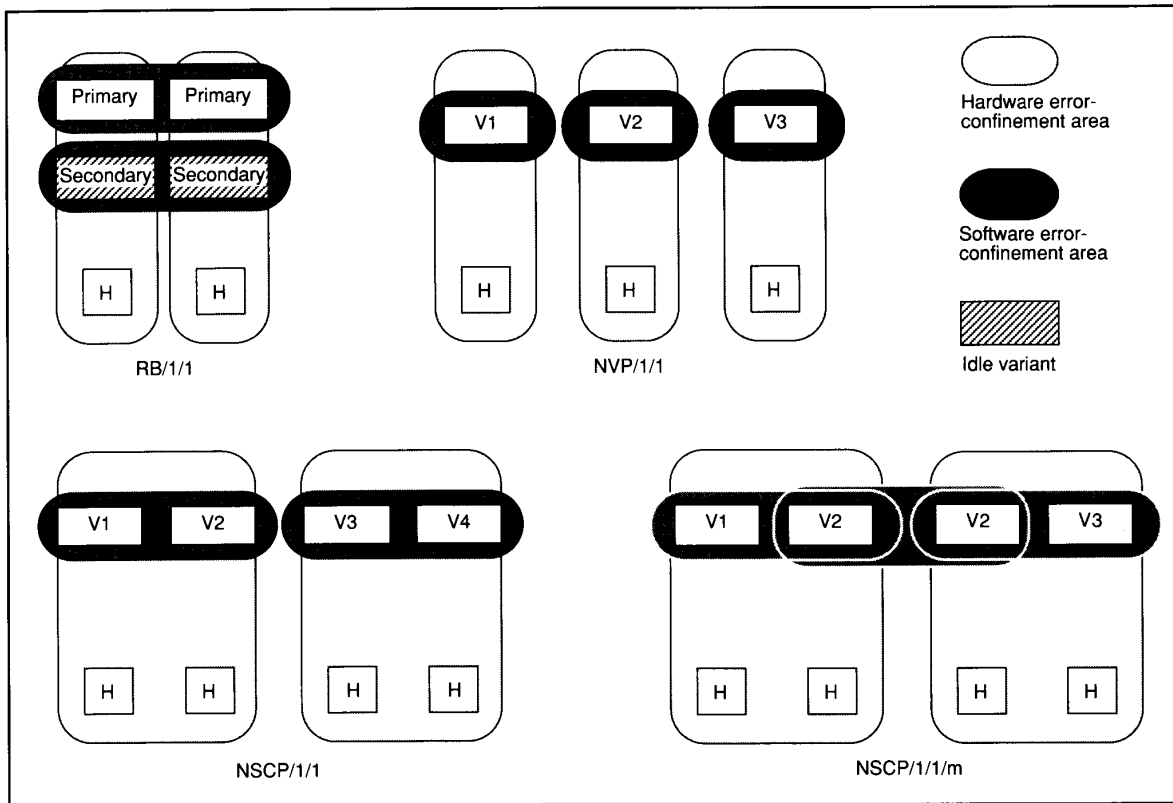
Figure 2. Architectures tolerating a single fault.

to the computation process, the states of distinct variants are different. Thus, in NSCP and NVP, when the variants execute in parallel on distinct (redundant) hardware, a given layer's diversity yields different states in its underlying layers, even if they are not diversified (except, of course, at the decision points). The decision to diversify layers underlying the application software involves additional considerations, such as determining the influence of those portions of the hardware and executive software specifically designed for the application, and determining how much confidence to place on experience validation for off-the-shelf components.

Structuring principles for architecture definition. Structuring is a prerequisite to mastering complexity, especially when dealing with fault tolerance.[5] Hardware-fault-tolerance mechanisms usually (and usefully) match the structuring of a system into layers.[10] Given performance considerations (that is, the time needed to recover from an error) and damage created

by error propagation, it is especially desirable that each layer have fault-tolerance mechanisms to process errors produced in that layer.

Implementing this principle in hardware to deal with software-fault tolerance requires that the redundant hardware components be in the same state when computation proceeds without error. Such a condition can be satisfied only if the variants execute sequentially, that is, in the RB approach. However, the diagnosis of hardware faults could be made possible by examining the syndrome provided by the deciders of the particular software-fault-tolerance method.

Another useful structuring mechanism is the error-confinement area,[10] a notion that cannot be considered separately from the architectural elements. The particular architectural elements we consider are:

• the hardware and associated executive software, which provide the necessary services for application software to execute (for concision, we call these

"hardware components"), and
• the variants of the application software.

Considering both hardware and software faults helps distinguish hardware and software error-confinement areas (HECAs and SECAs, respectively). In our discussion, a HECA covers at least one hardware component, and a SECA covers at least one software variant. Because of our definition of a hardware component, a HECA corresponds to that part of the architecture made passive after a solid hardware fault. It can thus be interpreted as a line replaceable unit.

Architectures tolerating a single fault. Three architectures correspond to the three software-fault-tolerance methods mentioned earlier. Figure 2 illustrates the SECA and HECA configurations for each method. The intersections of the SECAs and HECAs characterize the architectures' software- and hardware-fault-tolerance dependencies.

Table 3. Synthesis of the properties of the hardware-and-software-fault-tolerant architectures.

Architecture	Hardware Components/ Variants	Properties in Addition to Nominal Fault Tolerance		Fault-Tolerance After a HECA Is 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 two faults in hardware components of the same SECA; detection of three or four faults in hardware components	Tolerance of two independent faults in the same SECA; detection of two related faults in disjoint SECAs; detection of two, three, or four independent faults	Detection	Detection of independent faults
NSCP/1/1/m	4/3	Tolerance of two faults in hardware components of the same SECA	—	Detection	Detection of independent faults
NVP/1/1	3/3	Detection of two or three faults	Detection of two or three independent faults	Detection	Detection of independent faults
RB/2/1	3/2	Low error latency	—	Detection provided by local diagnosis	Tolerance of one independent fault
NSCP/2/1	6/3	Detection of three to six faults in hardware components	Detection of two or three independent faults	Detection	Detection of independent faults
NVP/2/1	4/3	Detection of three or four faults in hardware components; tolerance of combinations of single fault in hardware component and independent software fault in nonduplicated variant	Detection of two or three independent faults	Detection	Detection of independent faults
NVP/2/2	4/4	Detection of three or four faults in hardware components	Detection of two related faults; tolerance of two independent faults; detection of three or four independent faults	Detection	Detection of independent faults

We identify the architectures via a condensed expression of the form: $X/i/j/. . .$, where X is the software-fault-tolerance method (RB, NSCP, or NVP), i is the number of hardware faults tolerated, and j is the number of software faults tolerated. We add further labels to this expression when necessary. Table 3 summarizes the main fault-tolerance properties of the architectures discussed here and in the next section.

RB/1/1. This architecture duplicates a two-variant RB on two hardware components. Two variants and their instances of the acceptance test constitute two distinct SECAs and intersect each HECA. The RB method assures that each HECA is software-fault tolerant. A variant's independent faults are tolerated, while related faults between variants are detected.

However, related faults between a variant and the acceptance test cannot be tolerated or detected.

The hardware components operate in hot standby redundancy and always execute the same variant. Thus, hardware faults are detected by a high-coverage, concurrent comparison between the acceptance test results and the hardware results. When a discrepancy is detected during execution of the primary alternate or the acceptance test, the secondary executes so that the fault is tolerated (if the fault is soft). If the discrepancy persists (which would occur if the fault were solid), the failed HECA is identified by running diagnostic programs on each HECA. The failed HECA is thus made passive and service continuity is ensured.

The architecture remains software-fault tolerant after this hardware degradation,

and subsequent hardware faults are detected by either the acceptance test or periodic execution of the diagnostics.

NSCP/1/1. The basic NSCP/1/1 architecture (see Figure 2) comprises

- four hardware components grouped in two pairs in hot standby redundancy, each pair forming a HECA; and
- four variants grouped in two pairs, each pair forming a self-checking software component, with error detection performed by comparison. Each variant pair also forms a SECA associated with a HECA.

The computational states of the hardware components cannot be directly compared due to the diversification imposed by the variants. However, a comparison of each variant pair's results also effectively
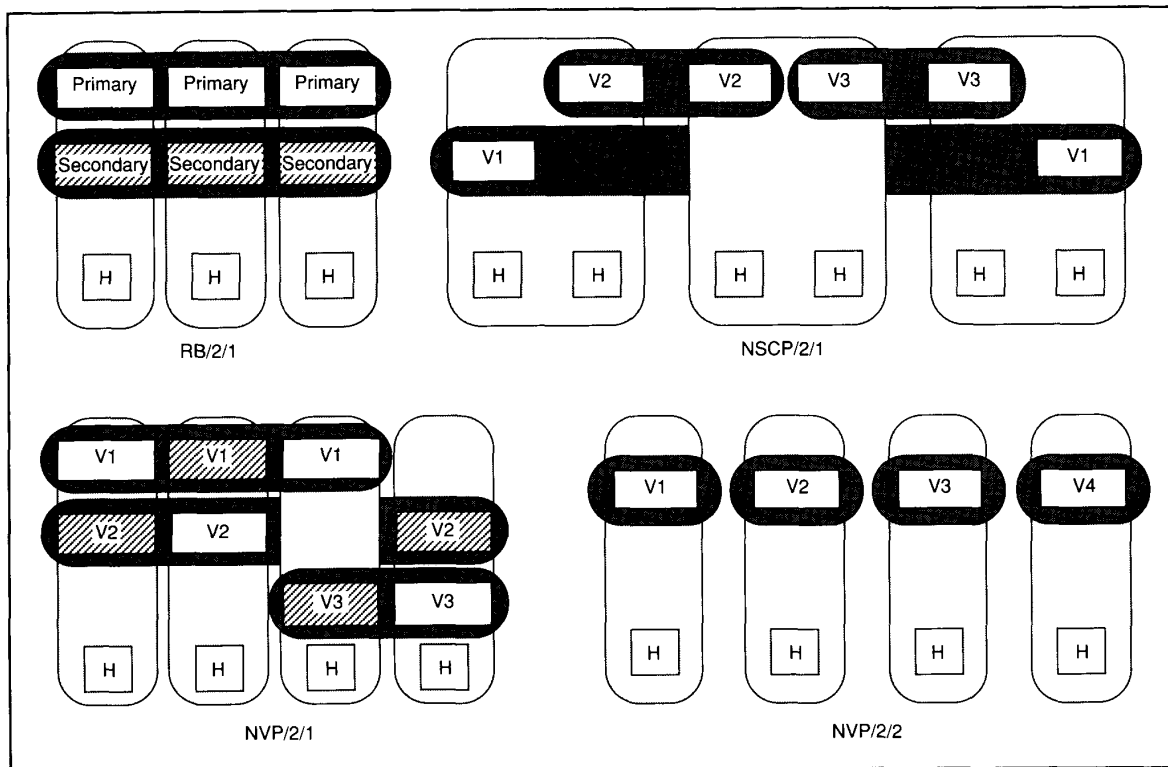
Figure 3. Architectures tolerating two consecutive faults.

compares the two hardware components in each HECA to check hardware faults (including design faults). Thus, a HECA is also a self-checking hardware component.

If the results from a HECA's variant pair differ, irrespective of the type of fault, then the results are delivered by the other HECA. If the discrepancy occurs repeatedly, thus indicating a solid hardware fault, then the HECA is made passive. The resulting degraded structure still allows detection of both software and hardware faults.

Besides nominally tolerating an independent software fault, the NSCP/1/1 architecture can also

- tolerate two simultaneous independent faults in a SECA,
- detect a related fault between two variants (each pertaining to one of the two disjoint SECAs), and
- detect three or four simultaneous independent software faults.

The NSCP/1/1 architecture corresponds to the architectural principle implemented in the Airbus A-320.[1] However, since requiring four variants would be prohibitive

in some applications, a modified architecture (NSCP/1/1/m) exists based on just three variants (see Figure 2).

To see the major difference in error processing between the NSCP/1/1 and NSCP/1/1/m architectures, consider a software fault in V2. Such a fault would cause a discrepancy in both self-checking components, implying an associated SECA covering all four software components and preventing any software-fault tolerance. Since this is the only event that can cause such an error syndrome (assuming a single independent fault), the "correct" result is immediately available as the one provided by V1 or V3. Hence, the NSCP/1/1/m architecture has a third SECA associated with V2 alone. However, the three additional fault-tolerance and detection capabilities of the NSCP/1/1 architecture listed above are lost.

NVP/1/1. The NVP/1/1 architecture is a direct implementation of the NVP method consisting of three hardware components, each running a distinct variant. The handling of both hardware faults (including design faults) and software faults is per-

formed at the software layer by the decider. In addition to tolerating an independent fault in a single variant, the architecture can detect independent faults in two or three variants.

The problem of discriminating between hardware and software faults, so that a hardware component is only made passive due to a solid fault, demonstrates the dependency between software- and hardware-fault tolerance. Because software faults are considered soft, a repeatedly disagreeing hardware component could easily be treated as a sign of a (solid) hardware fault. After the failed hardware component is made passive, the decider must be reconfigured as a comparator in case a hardware or software fault is subsequently activated.

Architectures tolerating two consecutive faults. Tolerance of two faults brings the distinction between soft and solid software faults into play. If the software faults are soft, then the number of variants is the same as in architectures that tolerate one fault. These architectures are of the type $X/2/1$. If the software faults are solid, then

the number of variants must increase because a failed variant cannot execute further. These architectures are of the type X/2/2.

Figure 3 shows architectures that tolerate two faults. The first three architectures (RB/2/1, NSCP/2/1, and NVP/2/1) tolerate two hardware faults and a single software fault. Another NVP-based architecture (NVP/2/2) deals with solid software faults by tolerating two consecutive (solid) faults in hardware or software.

RB/2/1. This architecture comprises three hardware components arranged in triple modular redundancy. Its ability to tolerate software faults is the same as that of RB/1/1. When a solid hardware fault is detected, the corresponding hardware component is made passive, thus degrading the architecture to a level analogous to the RB/1/1 architecture. Accordingly, each hardware component must include local diagnosis, even if it is basically useless in handling the first hardware fault.

NSCP/2/1. This architecture is a direct extension of NSCP/1/1/m. A supplementary duplex HECA supports a software self-checking component made up of two variants, resulting in a symmetric distribution of the three SECAs among the three HECAs. Since all the variants are duplicated, hardware faults can be instantly diagnosed by comparing the results from all hardware components. The architecture also detects simultaneous independent faults in two or three variants.

NVP/2/1. The NVP/2/1 architecture adds a hardware component to the NVP/1/1 architecture without introducing another variant. To maintain software-fault tolerance after a hardware component has been made passive, at least two instances of each variant must pertain to two distinct HECAs. Figure 3 shows only one of 18 possible configurations.

Of the two distinct variants associated with each HECA, one is active and the other is idle. At a given execution step, three hardware components execute three distinct variants, while the fourth hardware component executes a replica of one of the variants (V1 in this configuration). In addition to tolerating an independent software fault, this architecture can detect two or three simultaneous independent faults.

Tolerance of an independent fault is based on a vote incorporating the knowledge that two variants are identical. The
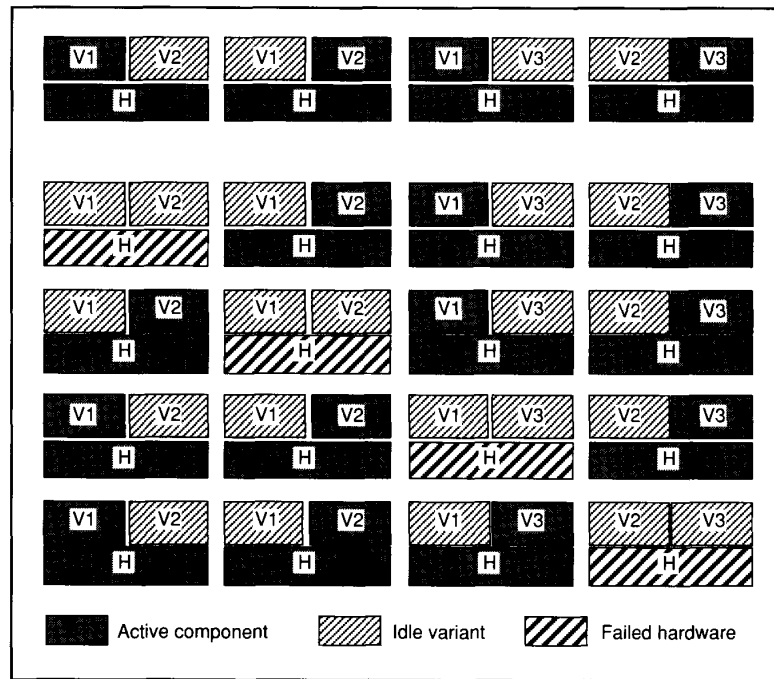

Figure 4. Various activations of the variants in the NVP/2/1 architecture.

unbalanced number of variant executions allows use of a double vote decision to improve the diagnosis of hardware faults (each vote includes the results of the nonduplicated variants and only one of the results of the duplicated variant). To understand this scheme, consider what happens when

- a hardware fault causes an error in one of the hardware components executing the duplicated variant (V1),
- a software fault causes an error in the duplicated variant,
- a hardware fault causes an error in one of the hardware components executing the nonduplicated variants (V2, V3), or a software fault causes an error in one of these variants.

In the first example, the fault is easily tolerated and diagnosed, since the three results agree on one vote and disagree on the second. Hence, the result of the duplicated variant is designated as false.

In the second example, the decider recognizes that the two votes are not unanimous and designates as false the results supplied by the duplicated variant. The "correct" result is thus immediately available as the one provided by the nonduplicated variants.

In the last example, tolerance is immediate, but the votes do not allow fault diagnosis. Since software faults are assumed to be nonrecurring, repeated failure of a hardware component leads to a diagnosis of a hardware fault. However, another form of diagnosis lets us relax this assumption: when a localized fault (attributable to one SECA or one HECA) occurs, the next execution step is performed after the active variants are reconfigured to match the duplicated variant with the affected HECA. The decider must then solve for one of the first two examples. A systematic rotation of the duplicated variants would also contribute to such a diagnosis.

After a failed hardware component is made passive, the active variants are reconfigured to distribute the SECAs among the remaining HECAs, forming disjoint areas. Figure 4 shows the distribution of active and idle variants among the three remaining HECAs after any of the HECAs have been made passive. In each case, the reconfiguration affects only a single HECA. Further decisions in this architecture are made by a vote among the active variants on the remaining HECAs. Thus, the degraded architecture is the same as the NVP/1/1 architecture.

Table 4. Probability of failure: $P_{S,X} = P_{S,D,X} + P_{S,U,X}$

	Probability of Detected Failure: $P_{S,D,X}$	Probability of Undetected Failure: $P_{S,U,X}$
RB/1/1	Separate: $(P_{I,RB})^2$ Common-mode: $P_{ID,RB} + P_{2V,RB}$	Common-mode: $P_{RVD,RB}$
NSCP/1/1	Separate: $4(P_{I,NSCP})^2 [1 - P_{I,NSCP} + ((P_{I,NSCP})^2/4)]$ Common-mode: $P_{ID,NSCP} + 4P_{2V,NSCP}$	Common-mode: $P_{2V,NSCP} + 4P_{3V,NSCP} + P_{4V,NSCP} + P_{RVD,NSCP}$
NVP/1/1	Separate: $3(P_{I,NVP})^2 [1-(2/3)P_{I,NVP}]$ Common-mode: $P_{ID,NVP}$	Common-mode: $3P_{2V,NVP} + P_{3V,NVP} + P_{RVD,NVP}$

Table 5. Comparison of analytical and experimental results.

$P_{I,NVP}$	$P_{2V,NVP}$	$P_{3V,NVP}$	$P_{S,NVP}$ (Table 4)	$P_{S,NVP}$ (Experimental)
2.91×10^{-3}	4.48×10^{-6}	1.09×10^{-6}	3.90×10^{-6}	3.67×10^{-6}

NVP/2/2. To understand the effect of solid software faults on architectures that tolerate two faults, consider the NVP method. Such an architecture requires four disjoint HECAs and SECAs, hence the NVP/2/2 architecture.

This architecture might seem to be a direct extension of NVP/1/1, adding only one HECA and an associated SECA, but there are major differences in error processing. The fault-tolerance decision is now based on finding a single set of two or more agreeing results among the four variant results provided. Also, after the first discrepancy is discovered, the designated hardware component and its associated variant are made passive without any attempt to diagnose the fault as a hardware or software fault. Further decisions are then

made by vote among the remaining variants, making the degraded architecture similar to the NVP/1/1 architecture. However, unlike the other NVP-based architectures, subsequent faults are treated the same as the first detected fault.

Besides tolerance to two consecutive independent software faults, this architecture lets the system tolerate two simultaneous independent faults, detect related faults between two variants, and detect simultaneous faults in three or four variants. The Fault-Tolerant Processor/ Attached Processor[4] is an implementation of this architecture: a quad configuration of the core fault-tolerant processor supports the execution of four different pieces of application software on four distinct application processors.
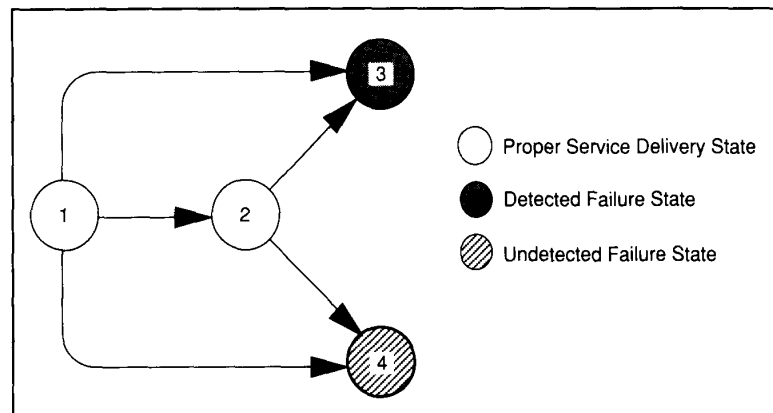
Analyzing and evaluating architectures

In discussing how to conduct a dependability analysis of hardware- and software-fault-tolerant architectures when adopting a Markov approach, we consider three architectures that tolerate a single hardware or software fault: RB/1/1, NSCP/1/1 and NVP/1/1.

Analyzing software-fault-tolerant architectures. Our analysis emphasizes the distinctions among the different sources of failures — independent and related faults in the variants and the decider — and assumes that only one type of fault can cause errors during each execution. Also, we do not address the underlying fault-tolerance mechanisms, that is, establishment and restoration of recovery points for RB, and version synchronization, establishment of cross-check points, and the decision mechanisms for NSCP and NVP.

We classify failures as separate or common-mode and as detected or undetected. Separate failures result from independent faults in the variants, whereas common-mode failures can result either from related faults or from independent faults in the decider. We also distinguish between two types of related faults: those among the variants and those between the variants and the decider. We consider a failure detected when the decider identifies no acceptable result and no output result is delivered. A failure is undetected when erroneous results are delivered.

We also assume that the probability of a fault is identical for all variants of a given architecture. We make this assumption to simplify the notation; it does not alter the significance of the results (it is simple to deduce the generalization to the case where variant characteristics are distinguished).

To characterize the probabilities of failure, we introduce the following notation for the $X/1/1$ architectures:

- $P_{I,X}$ is the probability of activating an independent fault in one variant of X on execution
- $P_{ID,X}$ is the probability of activating an independent fault in the decider of X on execution
- $P_{nV,X}$ is the probability of activating a related fault among n variants of X on execution
- $P_{RVD,X}$ is the probability of activating a

Figure 5. Generic model of architecture behavior.

○ Proper Service Delivery State

● Detected Failure State

▨ Undetected Failure State

Table 6. Specific state and transition definitions.

States and Interstate Transitions	RB/1/1	NSCP/1/1	NVP/1/1
1	2 (RB + hardware component) operational	2(2(variant + hardware component)) operational	3(variant + hardware component) operational
2	(RB + hardware component) operational	2(variant + hardware component) operational	2(variant + hardware component) operational
3	Detected failure	Detected failure	Detected failure
4	Undetected failure	Undetected failure	Undetected failure
1 to 2	Covered hardware component failure: $2c\lambda_{H,RB}$	Hardware component failure: $4\lambda_{H,NSCP}$	Hardware component failure: $3\lambda_{H,NVP}$
1 to 3	Noncovered hardware component failure or detected RB failure: $2\bar{c}\lambda_{H,RB} + \lambda_{S,D,RB}$	Detected NSCP failure: $\lambda_{S,D,NSCP}$	Detected NVP failure: $\lambda_{S,D,NVP}$
1 to 4	Undetected RB failure: $\lambda_{S,U,RB}$	Undetected NSCP failure: $\lambda_{S,U,NSCP}$	Undetected NVP failure: $\lambda_{S,U,NVP}$
2 to 3	Covered hardware component failure or detected RB failure: $c\lambda_{H,RB} + \lambda_{S,D,RB}$	Hardware component failure or detected two-variant failure: $2\lambda_{H,NSCP} + \lambda_{S,D,2V}$	Hardware component failure or detected two-variant failure: $2\lambda_{H,NVP} + \lambda_{S,D,2V}$
2 to 4	Noncovered hardware component failure or undetected RB failure: $\bar{c}\lambda_{H,RB} + \lambda_{S,U,RB}$	Undetected two-variant failure: $\lambda_{S,U,2V}$	Undetected two-variant failure: $\lambda_{S,U,2V}$

related fault among the variants and the decider of X on execution

• $P_{S,D,X}$ is the probability of a detected failure of X on execution

• $P_{S,U,X}$ is the probability of an undetected failure of X on execution

• $P_{S,D,X} + P_{S,U,X} = P_{S,X}$, the probability of a failure of X on execution

Table 4 summarizes the probabilities of failure and separates them into the separate/common-mode and detected/undetected categories. The table shows that either separate failures or common-mode failures can be detected, while only common-mode failures can go undetected. Comparing the probabilities is difficult due to the different parameter values for each architecture. However, some general observations are possible.

Although a large number of experiments have analyzed NVP, no quantitative study has reported the decider's reliability. Still, the probabilities of failure associated with the deciders can differ significantly. Due to the generic character and functional simplicity of the NSCP (comparison) and NVP (voting) deciders, the probabilities are likely ranked as follows:

$$P_{ID,NSCP} \leq P_{ID,NVP} \ll P_{ID,RB} \text{ and}$$
$$P_{RVD,NSCP} \leq P_{RVD,NVP} \ll P_{RVD,RB}$$

For separate failures, the influence of independent faults differs significantly. For RB, the probability of separate failure equals the square of the probability that independent faults will occur, while the probability for NVP is almost three times as much, and four times as much for NSCP. This difference results from the number of variants and the type of decision. However, it does not mean that RB and NSCP are the only architectures that allow detection of related-fault combinations among the variants. All related faults in NVP result in undetected failures. (This is due to our analysis' limitation to architectures that tolerate only one fault. Increasing the number of versions would allow NVP methods to detect some related faults.)

Although related faults among variants do not affect the probability of undetected failure for the RB architecture, they are the major contributor to undetected failure for NSCP and NVP. However, comparing the respective probabilities of undetected failure is not simple.

Experiments on multiversion software help us estimate some elementary probabilities of the expressions in Table 4. For example, Table 5 shows that the results obtained from the model agree with previous experimental results.[9] The first three

columns show the values derived for the model parameters from the experimental results. The fourth column gives the associated probability of failure, computed from the expressions of Table 4 using these parameter values and excluding decider fault parameters. The last column shows the experimental statistic for the probability of failure.

Evaluating hardware and software architectures. In modeling the behavior of the architectures, we assume

• only one type of fault can cause an error (either hardware or software) during each execution;

• the variant is not discarded after error detection and recovery, but is given the new input data at the next step (that is, software faults are soft); and

• the hardware components and the software-fault-tolerant architectures have constant failure rates.

Models. The generic model in Figure 5 describes the hardware and software behavior for the RB/1/1, NSCP/1/1, and NVP/1/1 architectures. Table 6 gives the specific definitions for states and interstate transitions of the model. The transition

Table 7. Time-dependent reliability and probability of undetected failure.

Method	Reliability	Probability of Undetected Failure
RB	$R_{RB}(t) \approx 1 - (2\bar{c}\lambda_{H,RB} + \lambda_{S,RB})t$	$P_{U,RB}(t) \approx \lambda_{S,U,RB}t$
NSCP	$R_{NSCP}(t) \approx 1 - \lambda_{S,NSCP}t$	$P_{U,NSCP}(t) \approx \lambda_{S,U,NSCP}t$
NVP	$R_{NVP}(t) \approx 1 - \lambda_{S,NVP}t$	$P_{U,NVP}(t) \approx \lambda_{S,U,NVP}t$

rates in Table 6 are based on the following notation:

- c is the hardware coverage factor of the RB/1/1 architecture, where $\bar{c} = 1-c$.
- $\lambda_{H,X}$ denotes the failure rate for a hardware component of the $X/1/1$ architecture.
- $\lambda_{S,D,X}$ and $\lambda_{S,U,X}$ denote the detected and undetected failure rates, respectively, for the fault-tolerant software X. If γ denotes the application software's execution rate, then we can express these failure rates as functions of the failure probabilities in Table 4: $\lambda_{S,D,X} = [P_{S,D,X}]\gamma$ and $\lambda_{S,U,X} = [P_{S,U,X}]\gamma$
- $\lambda_{S,D,2V}$ and $\lambda_{S,U,2V}$ denote the application software detected and undetected failure rates, respectively, of the NSCP/1/1 and NVP/1/1 architectures after an HECA has been made passive. These rates are defined as $[P_{S,D,2V}]\gamma$ and $[P_{S,U,2V}]\gamma$, respectively, where the probabilities of detected and undetected failure in the degraded two-version configuration are defined as $P_{S,D,2V} = 2 P_{I,2V} (1 - (P_{I,2V}/2)) + P_{ID,2V}$ and $P_{S,U,2V} = P_{RVD,2V}$

In RB/1/1, a hardware failure does not alter the architecture's software-fault tolerance, and a software failure does not alter its hardware-fault tolerance. We assume we can achieve near-perfect detection coverage for hardware faults, since both HECAs run the same variant simultaneously. Thus, the coverage considered here for the hardware-fault-tolerance mechanisms corresponds to local coverage, due to the diagnostic program and the acceptance test's capacity to identify hardware failures.

In NSCP/1/1, hardware- and software-fault-tolerance techniques are not independent, since the HECAs and the SECAs match. After a hardware component fails, the corresponding HECA and SECA are discarded. The resulting architecture comprises a pair of hardware components and a two-version software architecture, form-

ing a self-checking hardware and software architecture.

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

In both NSCP/1/1 and NVP/1/1, hardware faults are tolerated at the software layer through the decision algorithm (comparison or vote). Accordingly, only the software level accounts for the associated coverage, which the decider incorporates in the probability of a fault occurring. In the degraded NSCP/1/1 and NVP/1/1 architectures, the software is no longer fault-tolerant, so the variants' failure rates are important to the failure rate of the application software's degraded configuration.

Model processing. Combining the processing model in Figure 5 with the transition rates in Table 6 lets us derive the time-dependent probabilities of detected and undetected failure: $P_{D,X}(t)$ and $P_{U,X}(t)$, respectively, where t denotes time. In practice, we are interested mostly in the probability of undetected failure and in the reliability: $R_X(t) = 1 - [P_{D,X}(t) + P_{U,X}(t)]$. We can simplify these expressions for short missions (with respect to the mean times to failure). The simplified, approximate expressions (see Table 7) show that RB/1/1's reliability depends strongly on the coverage of fault diagnosis in the hardware components. Furthermore, the hardware failure rate is likely greater for RB/1/1 than for the other architectures, due to the extra memory needed to store the second variant. Also, to ensure near-perfect detection coverage, further hardware or software resources are needed to compare the results from each hardware processor, and storage is needed for the acceptance test and the diagnostic program.

The expressions also reveal that the application software's failure rate has an identical influence on the three architec-

tures, although this is tempered by the differences between the associated probabilities (identified in the section on analyzing software-fault-tolerant architectures).

The emergence of hardware-fault-tolerant commercial systems will increase the user's perception of the influence of design faults, due to these systems' tolerance of physical faults. Consequently, software-fault-tolerance schemes that use design diversity to give system users continuous service (as opposed to current implementations that preserve system core integrity through the termination of erroneous tasks[8]) are likely to spread from their current domain: safety-related systems. Accordingly, the approaches and results in this article are likely to apply more widely. ∎

References

1. U. Voges, ed., "Application of Design Diversity in Computerized Control Systems," *Proc. IFIP Workshop on Design Diversity in Action,* Springer-Verlag, Vienna, 1986.

2. J.-C. Laprie et al., "Hardware- and Software-Fault Tolerance: Definition and Analysis of Architectural Solutions," *Proc. 17th Int'l Symp. Fault-Tolerant Computing,* 1987, Computer Society Press, Los Alamitos, Calif., Order No. 778 (microfiche only), pp. 116-121.

3. 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 Trans. Computers,* Vol. 38, No. 5, May 1989, pp. 626-636.

4. J.H. Lala and L.S. Alger, "Hardware- and Software-Fault Tolerance: A Unified Archi-

tectural Approach," *Proc. 18th Int'l Symp. Fault-Tolerant Computing*, 1988, Computer Society Press, Los Alamitos, Calif., Order No. 867, pp. 240-245.

5. B. Randell, "Design-Fault Tolerance," in *The Evolution of Fault-Tolerant Computing*, A. Avizienis, H. Kopetz, and J.-C. Laprie, eds., Springer-Verlag, Vienna, 1987, pp. 251-270.

6. A. Avizienis, "The N-Version Approach to Fault-Tolerant Systems," *IEEE Trans. Soft-*

ware Engineering, Vol. SE-11, No. 12, Dec. 1985, pp. 1,491-1,501.

7. S.S. Yau and R.C. Cheung, "Design of Self-Checking Software," *Proc. 1975 Int'l Conf. Reliable Software*, pp. 450-457.

8. J.N. Gray, "Why Do Computers Stop and What Can Be Done About It?" *Proc. Fifth Symp. Reliability in Distributed Software and Database Systems*, 1986, Computer Society Press, Los Alamitos, Calif., Order No. 690 (microfiche only), pp. 3-12.

9. J.C. Knight and N.G. Leveson, "An Empirical Study of Failure Probabilities in Multiversion Software," *Proc. 16th IEEE Int'l Symp. Fault-Tolerant Computing*, 1986, Computer Society Press, Los Alamitos, Calif., Order No. 703, pp. 165-170.

10. D.P. Siewiorek and D. Johnson, "A Design Methodology for High-Reliability Systems: The Intel 432," in *The Theory and Practice of Reliable System Design*, D.P. Siewiorek and R.S. Swarz, eds., Digital Press, 1982.

Jean-Claude Laprie is directeur de recherche of CNRS, the French national organization for scientific research. He joined LAAS-CNRS in 1968, where he has directed the research group on fault tolerance and dependable computing since 1975. He was chair of the IEEE Computer Society Technical Committee on Fault-Tolerant Computing in 1984-1985 and has been chair of the IFIP working group on Dependable Computing and Fault Tolerance since 1986.

Laprie received the Certified Engineer degree from the Higher National School for Aeronautical Constructions, Toulouse, France, in 1968, and the Doctor in Engineering degree in automatic control and the Doctor-ès-Sciences degree in computer science from the University of Toulouse in 1971 and 1975, respectively.

Christian Béounes is chargé de recherche of INRIA, the French National Institute for Computing and Automatic Control Research. He joined LAAS in 1974 as a member of the group on fault tolerance and dependable computing. His research interests include stochastic petri nets, modeling, and dependability evaluation.

Béounes received the Certified Engineer degree from the National Institute of Applied Sciences, Toulouse, France, in 1973 and the Doctor in Engineering degree in automatic control from the University of Toulouse, in 1977.

Karama Kanoun is chargée de recherche of CNRS. She joined LAAS in 1977 as a member of the group on fault tolerance and dependable computing. Her research interests include modeling and evaluating computer system dependability, considering hardware as well as software.

Kanoun received the Certified Engineer degree from the National School of Civil Aviation, Toulouse, France in 1977, and the Doctor-Engineer and Doctor-ès-Sciences degrees from the National Institute Polytechnique of Toulouse in 1980 and 1989, respectively.

Readers can contact the authors at LAAS-CNRS, 7 Avenue du Colonel Roche, 31077 Toulouse Cedex, France.

Jean Arlat is chargé de recherche of CNRS. He joined LAAS-CNRS, where he is a member of the research group on fault tolerance and dependable computing, in 1976. His research focuses on evaluating dependability including both analytical modeling and experimental fault injection.

Arlat received the Certified Engineer degree from the National Institute of Applied Sciences, Toulouse, France and the Doctor in Engineering degree from the National Polytechnic Institute, Toulouse, France, in 1976 and 1979, respectively. He is a member of the IEEE Computer Society's technical committees on fault tolerance and simulation.