3

Design Methods, Programming Techniques, and Issues

Developing dependable, critical applications is not an easy task. The trend toward increasing complexity and size, distribution on heterogeneous platforms, diverse accidental and malicious origins of system failures, the consequences of failures, and the severity of those consequences combine to thwart the best human efforts at developing these applications. In this chapter, we will examine some of the problems and issues that most, if not all, software fault tolerance techniques face. (Issues related to specific techniques are discussed in Chapters 4 through 6 along with the associated technique.) After examining some of the problems and issues, we describe programming or implementation methods used by several techniques: assertions, checkpointing, and atomic actions. To assist in the design and development of critical, fault-tolerant software systems, we then provide design hints and tips, and describe a development model for dependable systems and a design paradigm specific to *N*-version programming (NVP).

3.1 Problems and Issues

The advantages of software fault tolerance are not without their attendant disadvantages, issues, and costs. In this section, we examine these issues and potential problems: similar errors, the consistent comparison problem (CCP), the domino effect, and overhead. These are the issues common to

many types of software fault tolerance techniques. Issues that are specific to individual techniques are discussed in Chapters 4 through 6, along with the associated technique. Knowing the existence of these problems and understanding the problems may help the developer avoid their effects or at least understand the limitations of the techniques so that knowledgeable choices can be made.

3.1.1 Similar Errors and a Lack of Diversity

As stated in the introductory chapter, the type of software fault tolerance examined in this book is *application* fault tolerance. The faults to be tolerated arise from software design and implementation errors. These cannot be detected by simple replication of the software because such faults will be the same in all replicated copies—hence the need for diversity. (We discussed the need for and experiments on diversity in Chapter 2.) Diversity allows us to be able to detect faults using multiple versions of software and an adjudicator (see Chapter 7). In this section, we examine the faults arising from a lack of adequate diversity in the variants used in design diverse software fault tolerance techniques and the problems resulting from a lack of diversity.

One of the fundamental premises of the NVP software fault tolerance technique (described in Section 4.2) and other design diverse techniques, especially forward recovery ones, is that the lack of "independence of programming efforts will assure that residual software design faults will lead to an erroneous decision by causing similar errors to occur at the same [decision point]" [1] in two or more versions. Another major observation is that "[NVP's] success as a method for run-time tolerance of software faults depends on whether the residual software faults in each version are distinguishable" [2, 3]. The reason errors need to be distinguishable is because of the adjudicator—forward recovery design diverse techniques typically use some type of voter to decide upon or adjudicate the correct result from the results obtained from the variants. (Adjudicators are discussed in Chapter 7.)

The use of floating-point arithmetic (FPA) in general computing produces a result that is accurate only within a certain range. The use of design diversity can also produce individual variant results that differ within a certain range, especially if FPA is used. A tolerance is a variance allowed by a decision algorithm. Two or more results that are approximately equal within a specified tolerance are called *similar results*. Whether the results are correct or incorrect, a decision algorithm that allows that tolerance will view the similar results as correct. Two or more similar results that are erroneous are referred to as *similar errors* [1, 4], also called identical and wrong answers

(IAW). If the variants (functionally equivalent components) fail on the same input case, then a *coincident failure* [5] is said to have occurred. If the actual, measured probability of coincident variant failures is significantly different from what would be expected by chance occurrence of these failures (assuming failure independence), then the observed coincident failures are *correlated* or dependent [6–9].

When two or more correct answers exist for the same problem, for the same input, then we have multiple correct results (MCR) [10, 11]. An example of MCR is finding the roots of an *n*th order equation, which has *n* different correct answers. The current algorithms for finding these roots often converge to different roots, and even the same algorithm may find different roots if the search is started from different points. Figure 3.1 presents a taxonomy of variant results, the type of error they may indicate, the type of

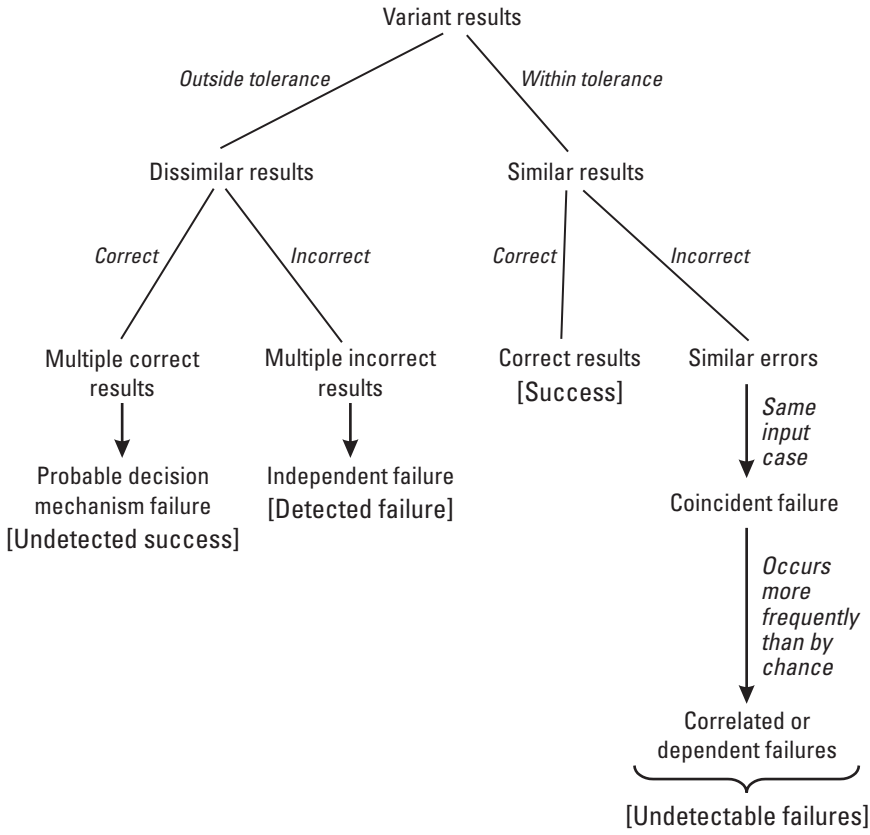

Figure 3.1 A taxonomy of variant results.

failure the error may invoke, and the resulting success or failure detected. The arrows show the errors causing the failures to which they point.

Figure 3.2 illustrates some of these errors and why they pose problems for fault-tolerant software. In this example, the same input, *A*, is provided to each variant. Variants 1 and 2 produce results, r_1 and r_2, respectively, that are within a predefined tolerance of each other. Suppose a majority voter-type decision mechanism (DM) is being used. Then, the result returned by the decision mechanism, $r*$, is equal to r_1 or r_2 (or some combination of r_1 and r_2 such as an average, depending on the specific decision algorithm). If r_1 and r_2 are correct, then the system continues this pass without failure. However, if r_1 and r_2 are erroneous, then we have similar errors (or IAW answers) and an incorrect result will be returned as the "valid" result of the fault-tolerant subsystem. Since variants 1 and 2 received the same input, *A*, we also have a coincident failure (assuming a failure in our example results from the inability to produce a correct result). With the information given in this example, we cannot determine if correlated or dependent failures have occurred. This example has illustrated the havoc that similar errors can play with multiversion software fault tolerance techniques.

3.1.2 Consistent Comparison Problem

Another fundamental problem is the CCP, which limits the generality of the voting approach for error detection. The CCP [12, 13] occurs as a result of

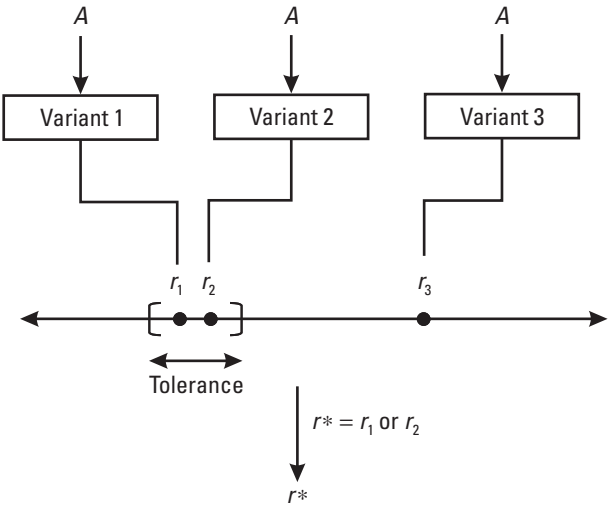

Figure 3.2 Example of similar results.

finite-precision arithmetic and different paths taken by the variants based on specification-required computations. Informally stated, "the difficulty is that if N versions operate independently, then whenever the specification requires that they perform a comparison, it is not possible to guarantee that the versions arrive at the same decision, i.e., make comparisons that are consistent" [14]. These isolated comparisons can lead to output values that are completely different rather than values that differ by a small tolerance. This is illustrated in Figure 3.3. The following example is from [12].

Suppose the application is a system in which the specification requires that the actions of the system depend upon quantities, x, that are measured by sensors. The values used within a variant may be the result of extensive computation on the sensor measurements. Suppose such an application is

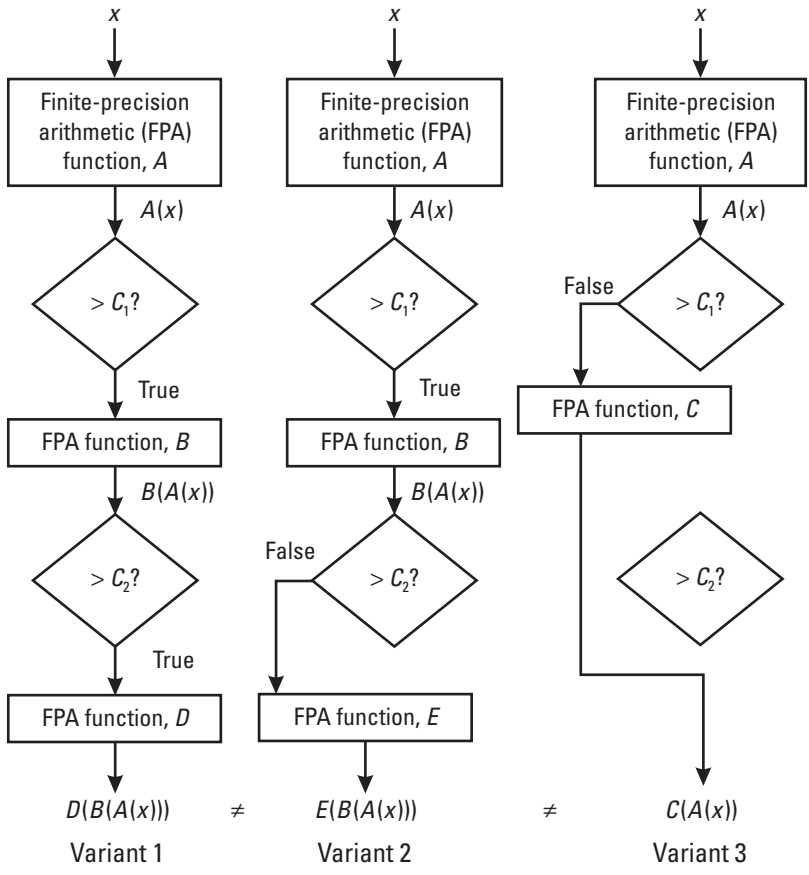

Figure 3.3 Consistent comparison problem yields variant result disagreement.

implemented using a three-variant software system and that at some point within the computation, an intermediate quantity, $A(x)$, has to be compared with an application-specific constant C_1 to determine the required processing. Because of finite-precision arithmetic, the three variants will likely have slightly different values for the computed intermediate result. If these intermediate result values are very close to C_1, then it is possible that their relationships to C_1 are different. Suppose that two of the values are less than C_1 and the third is greater than C_1. If the variants base their execution flow on the relationships between the intermediate values and C_1, then two will follow one path and the third a different path. These differences in execution paths may cause the third variant to send the decision algorithm a final result that differs substantially from the other two, $B(A(x))$ and $C(A(x))$.

It may be argued that the difference is irrelevant because at least two variants will agree, and, since the intermediate results were very close to C_1, either of the two possible results would be satisfactory for the application. If only a single comparison is involved, this is correct. However, suppose that a comparison with another intermediate value is required by the application. Let the constant involved in this decision be C_2. Only two of the variants will arrive at the comparison with C_2 (since they took the same path after comparison with C_1). Suppose that the intermediate values computed by these two variants base their control flow on this comparison with C_2, then again their behavior will differ. The effect of the two comparisons, one with each constant, is that all variants might take different paths and obtain three completely different final results, for example, $D(B(A(x)))$, $E(B(A(x)))$, and $C(A(x))$. All of the results are likely to be acceptable to the application, but it might not be possible for the decision algorithm to select a single correct output. The order of the comparisons is irrelevant, in fact, since different orders of operation are likely if the variants were developed independently. The problem is also not limited to comparison with constants because if two floating-point numbers are compared, it is the same as comparing their differences with zero.

The problem does not lie in the application itself, but in the specification. Specifications do not (and probably cannot) describe required results down to the bit level for every computation and every input to every computation. This level of detail is necessary, however, if the specification is to describe a function in which one, and only one, output is valid for every input [15]. It has been shown that, without communication between the variants, there is no solution to the CCP [12].

Since the CCP does not result from software faults, an n-version system built from fault-free variants may have a nonzero probability of being unable

to reach a consensus. Hence, if not avoided, the CCP may cause failures to occur that would not have occurred in non-fault-tolerant systems. The CCP has been observed in several NVP experiments. There is no way of estimating the probability of such failures in general, but the failure probability will depend heavily on the application and its implementation [14]. Although this failure probability may be small, such causes of failure need to be taken into account in estimating the reliability of NVP, especially for critical applications.

Brilliant, Knight, and Leveson [12] provide the following formal definition of CCP:

> Suppose that each of N programs has computed a value. Assuming that the computed values differ by less than ε ($\varepsilon > 0$) and that the programs do not communicate, the programs must obtain the same order relationship when comparing their computed value with any given constant.

Approximate comparison and rounding are not solutions to this problem. Approximate comparison regards two numbers as equal if they differ by less than a tolerance δ [16]. It is not a solution because the problem arises again with $C + \delta$ (where C is a constant against which values are compared). Impractical avoidance techniques include random selection of a result, exact arithmetic, and the use of cross-check points (to force agreement among variants on their floating-point values before any comparisons are made that involve the values).

When two variants compare their computed values with a constant, the two computed values must be identical in order for the variants to obtain the same order relationship. To solve the CCP, an algorithm is needed that can be applied independently by each correct variant to transform its computed value to the same representation as all other correct variants [12]. No matter how close the values are to each other, their relationships to the constant may still be different. The algorithm must operate with a single value and no communication between variants to exchange values can occur since these are values produced by intermediate computation and are not final outputs. As shown by the following theorem, there is no such algorithm, and hence, no solution to the CCP [12].

> Other than the trivial mapping to a predefined constant, no algorithm exists which, when applied to each of two n-bit integers that differ by less than $2k$, will map them to the same m-bit representation ($m + k \leq n$).

In the investigation of practical avoidance techniques for the CCP, the major characteristic that needs to be considered is whether or not the application has state information that is maintained from frame to frame, that is, whether or not the application maintains its history [12]. Systems and associated CCP avoidance techniques can be characterized as shown in Figure 3.4. Each of these types of systems and the avoidance technique proposed by Brilliant, Knight, and Leveson [12] are discussed in the following paragraphs. The immediate effect of inconsistent comparison on a system is that a consensus might not be reached. The extent of the resulting damage varies with the application and has a substantial impact on the effectiveness of measures designed to handle the damage [12]. The avoidance approach requires that enhancements be made to the implementation of an NVP system.

3.1.2.1 Systems with No History

Some simple control systems have no history and thus compute their outputs for a given frame using only constants and the inputs for that frame. If no consensus is reached in one frame and if the inputs are changing, then it is extremely unlikely that the lack of consensus will last for more than a short time. After a brief interval, the inputs should leave the region of difficulty. Doing so, subsequent comparisons will be consistent among the variants. Hence, the effects of the CCP in systems with no history are transient.

An avoidance approach, using confidence signals, for the CCP in systems with no history is described in [12]. Each variant determines, for itself, whether the values used in comparisons were close enough to warrant

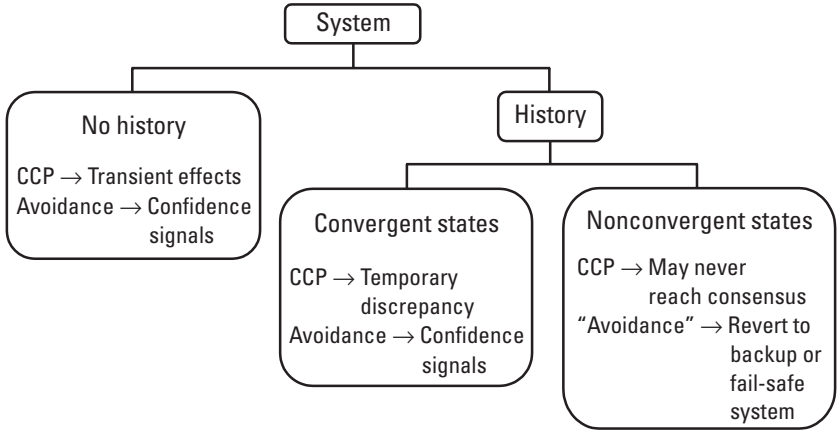

Figure 3.4 Consistent comparison problem avoidance techniques depend on system history maintenance.

suspicion of inconsistency. If a possibly inconsistent solution is detected by the variant, it signals the voter of the event. The voter is unable to tell the difference between the occurrence of an inconsistent comparison and a failed variant, so it ignores the flagged variants' results. The voter can then vote using the results from the variants that indicated confidence in their results. Hence, the fault tolerance may be reduced or even eliminated in this situation. System recovery under these circumstances is application dependent, but it may be possible to treat the situation as a single-cycle failure [12]. This approach requires fairly extensive modifications to the system structure. For example, each variant result would have to be supplemented by a confidence signal, and the voter would have to be modified to incorporate these signals into its decision-making logic.

3.1.2.2 Systems with Convergent States

The situation is much more complex for systems with history, that is, those that maintain internal state information over time. In these systems, the failure to reach a consensus may coincide with differences in the internal state information among the variants [12]. The duration of these internal state differences varies among applications.

In some applications, the state information is revised with the passage of time and, once the inputs have changed so that comparisons are again consistent, the variants may revise their states to also be consistent. In these systems with convergent states, the entire system is once again consistent and operation can safely proceed. An example [12] of this type of application is an avionics system in which the flight mode is maintained as internal state information. If the flight mode is determined by height above ground, then if a measurement is taken that is close to the value at which the mode is changed, different variants might reach different conclusions about which mode to enter. If the variants continue to monitor the height sensor, any inconsistency that occurs should be rapidly corrected.

Inconsistent comparisons may cause a temporary discrepancy among variant states in systems with convergent states. A confidence signal approach may also be used with these systems [12]. Each variant must maintain confidence information as part of its state. If a part of the system's state information is based on a comparison that may be inconsistent, then the variant must indicate a "No confidence" signal to the voter for its results. The no confidence state for this variant remains until the system state is reevaluated. The time required to reevaluate the state is application dependent. During the reevaluation period the system is not fault tolerant. In addition, the time to reevaluate the state may be unacceptably long.

3.1.2.3 Systems with Nonconvergent States

Other applications (i.e., systems with nonconvergent states) determine and then never reevaluate some state information. An example [12] of this type system is sensor processing in which one variant may determine that a sensor has failed and subsequently ignore it. Other variants may not make the same decision at the same point in time, and, depending on subsequent sensor behavior, may never conclude that the sensor has failed. Hence, although the inputs change, the variants may continue to arrive at different correct outputs long after comparisons become consistent because the sets of state information maintained by the individual variants are different.

Once the variants in a system with nonconvergent states acquire different states, the inconsistency may persist indefinitely. Though no variant has failed, the variants may continue to produce different outputs. In the worst case, the NVP system may never again reach a consensus on a vote. There is no simple avoidance technique that can be used for systems with nonconvergent states. The only practical approach in systems of this type seems to be to revert to a backup or a fail-safe system [12].

3.1.3 Domino Effect

While the CCP of the previous section can generally affect design diverse forward recovery software fault tolerance techniques, the domino effect discussed here can generally affect backward recovery techniques. The *domino effect* [17] refers to the successive rolling back of communicating processes when a failure is detected in any one of the processes.

To implement software fault tolerance in concurrent systems (of multiple cooperating processes that communicate with each other via messages), one cannot simply apply some fault tolerance technique in each separate process. If this is done, then each process will have its own error detection mechanism and would establish its own recovery point(s). When one process detects an error and attempts recovery to its recovery point or checkpoint, this can result in an inconsistent global system state unless the other relevant processes are also rolled back. When rolling back the faulty process to its recovery point, the messages issued by that process may also be faulty, so they must be recalled [17, 18]. This recall will force the other processes to roll back to their recovery points that precede receipt of the recalled messages. This recovery and recall continues until the system reaches a stable state, which may be the initial state. This continual rollback and recall is the domino effect, resulting when recovery and communication operations are not

coordinated. This causes the loss of the entire computation that was performed prior to the detection of the initial error.

An example will help illustrate the domino effect (see Figure 3.5). (Similar examples are provided in [18–21] and others. The example provided here is derived from [22].) In the figure below, the communicating processes are labeled P1 and P2. At time T1, P1 detects an error and must roll back to recovery point R6. Because of the communications, C5, between P1 and P2, process P2 has to roll back to its recovery point R5. Because of this rollback, the effect of C4 has to be removed, so P1 has to roll back to R4. Because of C3, P2 has to roll back to R3. Because of C2, P1 has to roll back to R1 and because of C1, P2 has to roll back to R2. Now both processes have rolled back to their initial state at T0 and lost the computations performed between T0 and T1.

The avoidance of the uncontrolled rolling back evidenced by the domino effect is achieved if system consistent states, which serve as recovery points, can be established. A consistent state of a system conforms to the system's correctly reachable states and the events history as reflected in the system behavior (its interface) [23]. A consistent state allows the system to achieve an error-free state that leads to no contradictions and conflicts within the system and its interfaces. All communications between processes and their order of occurrence are taken into account. To support consistency, some restrictions on the communication system must be enforced [23]:

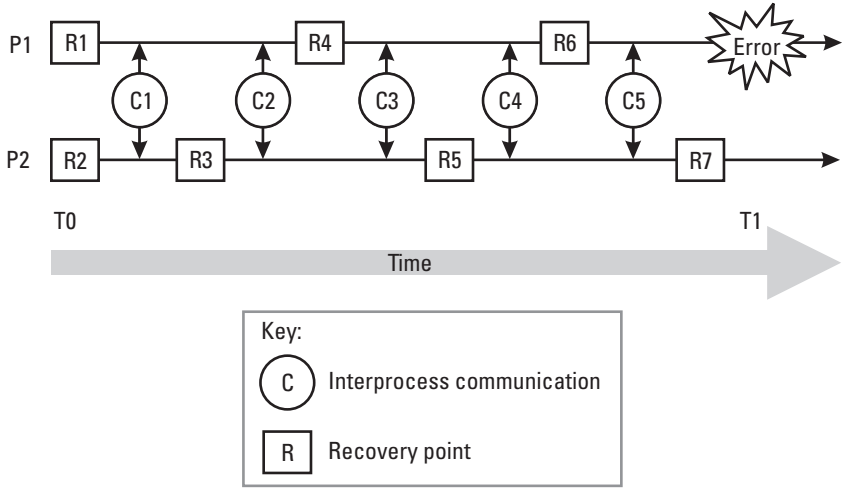

Figure 3.5 The domino effect. (*Source:* [22], © 1991, IEEE. Reprinted with permission.)

- Communication delay is negligible and can be considered zero.
- Communication maintains a partial order of data transfer. All messages sent between a particular pair of processes are received at the destination in the order they were sent.

Consistent states can be determined statically or dynamically. The static approach is a language-based approach in which the consistent state is determined at compile time. At compile time a recovery line is set comprising a set of recovery points, one for each process, to which the processes will roll back. The conversation scheme [17, 24] is a well-known static approach and the oldest approach for overcoming the domino effect. In the conversation scheme, processes that are members of a conversation may communicate with each other, but not with processes outside the conversation. The processes must establish a recovery point when they enter a conversation, and all processes must leave the conversation together. This technique is discussed more in Chapter 4.

The dynamic approach uses stored information about communication and recovery points to set up a recovery line only after an error occurs. The programmer-transparent coordination scheme [18, 25, 26] is a dynamic approach that overcomes the domino effect by relying on an intelligent underlying machine. Detailed implementations of models and recovery protocols based on state descriptions can be found in the literature, such as in [27].

3.1.4 Overhead

The benefits of software fault tolerance do not come without a price. In this section, we examine the overhead incurred in the use of software fault tolerance in terms of space (memory), time, and cost. Given this information (and information specific to individual techniques presented in Chapter 4), one can make a more informed decision on the use of software fault tolerance.

Table 3.1 [28] provides a summary of the overhead involved in software fault tolerance for tolerating one fault. Overhead is described in terms of both structural overhead and operational time overhead. The table does not include overhead that is common to all approaches (including that overhead that should be present in non-fault-tolerant software) such as checking input variables to ensure their values are within a valid range or checking for results that are obviously grossly wrong. For example, the recovery block

Table 3.1
Software Fault Tolerance Overhead for Tolerating One Fault (*From:* [28], © 1995 John Wiley & Sons, Ltd. Reproduced with permission.)

Method Name		**Structural Overhead**		**Operational Time Overhead**		
		Diversified Software Layer	Mechanisms (Layers Supporting the Diversified Software Layer)	Systematic		On Error Occurrence
				Decider	Variants Execution	
RcB		One variant and one AT	Recovery cache	Acceptance test execution	Accesses to recovery cache	One variant and AT execution
NSCP	Error detection by ATs	One variant and two ATs	Result switching		Input data consistency and variants execution synchronization	Possible result switching
	Error detection by comparison	Three variants	Comparators and result switching	Comparison execution		
NVP		Two variants	Voters	Vote execution		Usually neglectable

(RcB) technique includes structural overhead for one variant, an acceptance test (AT), and its recovery cache and operational time overhead for executing the AT, accessing the recovery cache, and when an error is detected, executing an additional variant and the AT on the second variant's results. We hold further discussion of the details of this table for the technique discussions of Chapters 4 through 6. It is provided here to briefly illustrate some of the non-cost-related overhead.

As discussed in Chapter 2, all the software fault tolerance techniques require diversity in some form and this diversity in turn requires additional space or time, or both. Xu, Bondavalli, and Di Giandomenico [29] provide an illustration (see Figure 3.6) that summarizes the space and time overheads of software fault tolerance techniques. Space is defined here as the amount of hardware, such as the number of processors, needed to support parallel execution of multiple variants. Time is defined for the figure as the physical time required to execute the variants sequentially. For example, the NVP technique requires additional space for its n variants, so it is to the upper (right) side of the space continuum on the figure. It is also on the lower (top) side of the time continuum because all the variants are executed in parallel. (Xu, Bondavalli, and Di Giandomenico [29] developed a technique, self-configuring optimal programming (SCOP), presented in Chapter 6, that attempts to optimize

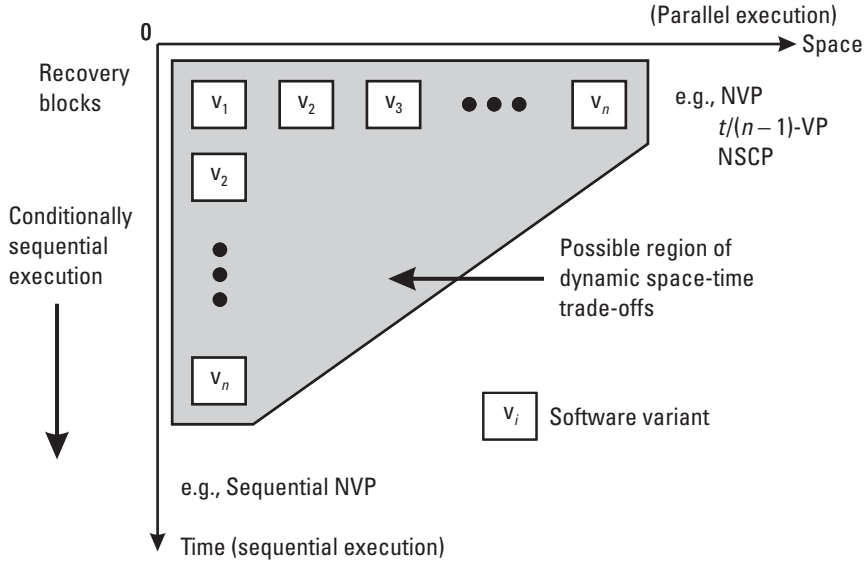

Figure 3.6 Space and time redundancy in software fault tolerance. (*Source:* [29], © 1995, Springer-Verlag, Figure 1, p. 158.)

these overheads.) For use here, the figure provides a basis for understanding the space and time requirements of, and possible trade-offs between, the software fault tolerance techniques being considered for use.

Software fault tolerance also introduces additional overhead in terms of costs. The cost-effectiveness of software has been the subject of debate for many years. The question usually posed is this: Is it better to devote the extra effort to develop the additional variants for diverse software or to devote that effort to the verification and validation (V&V) of one "really good" piece of software? (Halton [30] provides an interesting discussion of this question, favoring diversity.) Below we provide brief descriptions of experimental results on the cost of diversity, then continue with more specific cost information.

The following summaries provide brief overviews of experimental results on the costs of diversity. Note that, in general, the results indicate that the cost of threefold diversity (e.g., NVP with $n = 3$) is not three times that of a single development (it is less) and the cost of twofold diversity is less than twice that of a single development. In addition, not all parts of the software's functionality are critical; that is, only a small part of the software may need to be made fault tolerant. Software fault tolerance may also be less expensive than alternative means of assurance. When examining the cost of software fault tolerance, it is useful to keep the focus on the cost implications on the overall project.

- Several experiments, for example [31–33], have shown that the development and maintenance costs for three-variant diversity can be twice that of a single development and less than double for two-variant diversity.

- The Ericsson company [32] found that, for their railway interlocking system, the costs of developing two functionally equivalent software variants is not double the cost of a single variant because: (a) not all parts of the system are critical, hence only the critical parts may require fault tolerance; (b) while the cost of program specification (design), coding, and testing is doubled, the cost of requirement specification, system specification, test specification, and system test execution is not doubled. In the Ericsson system [32], the majority of the system is concerned with complex computations to make train control decisions. The control commands are submitted to the interlocking software and only the interlocking software is diversely implemented.

- Panzl [34] found that two-variant development with back-to-back (comparison) testing increases the initial development cost by 77% (not a factor of 2), but reduced the number of residual errors from 69 to 2.
- Laprie, et al. [35] analyzed the cost of several software fault tolerance techniques and found that in a multiple variant software development project typical component cost is about 75% to 80% of single-variant costs.
- An experiment at the University of Newcastle upon Tyne estimated the RcB technique's overhead for two variants to be 60% [36] (i.e., 0.8 times the cost of a non-fault-tolerant variant, a cost increase factor of 1.6).
- Another experiment estimated the cost of NVP for $n = 3$ variants at 2.26 times the cost of a one-variant program [37] (and a cost increase factor of 1.5 for $n = 2$). Hence, the cost of a variant in NVP was evaluated as 0.75 times the cost of a non-fault-tolerant variant.
- There are a number of models of the cost of fault-tolerant software [35, 38–43]. We will examine one of these in detail later in this section.
- Some industries, such as aircraft and nuclear power, are subject to official regulation. The costs of demonstrating safety in these industries can far outweigh the development costs. For example, over 500 person-years of effort in safety assessment [44, 45] (not including lost income associated with licensing delays) have been used in the Sizewell B nuclear power station computer-based protection system. In some cases, diversity (not necessarily *software* diversity) has been used to make a more convincing safety case.
- Acceptable alternatives to software fault tolerance may also increase the system's cost. For example, formal methods can be used to prove that a program meets its specification (see Section 1.3.1.3). However, there are high costs associated with these alternatives and there remains the risk of faults within the process and results of these approaches.

Kanoun [46] provides the results of examining working hours recorded during the development of a real-life software system (composed of two self-checking diverse variants). Kanoun evaluated the cost overhead induced by the development of the second variant with respect to the cost of the

principal variant, which was considered as a reference. The results indicate this overhead varies from 15% to 134% according to the development phase (134% for coding and unit tests together; about 100% for integration tests, general tests, maintenance, and analysis; about 60% for specification analysis, design, and documentation; and 25% for functional specifications). The average was around 64% for all phases excluding the effort spent for requirement specifications and system tests. The average is between 42% and 71% if the requirement specification phase only is excluded (assuming the system test overhead is 0% and 100%, respectively). Kanoun's results confirm those published in previous work (see above). The results are especially valuable since they were observed for a real system in an industrial environment (versus an experimental environment).

We will examine one of the cost models mentioned above in more detail. Laprie, et al. [35] examined the costs of several software fault tolerance techniques and provided a model for determining the cost of fault-tolerant software with respect to the cost of non-fault-tolerant software. Laprie starts with a cost distribution across life cycle phases for classical, non-fault-tolerant software (see Table 3.2, with these entries based on [47]). Since fault-tolerant software is used mainly with critical applications and the cost distribution is based on no specific class of software, [35] provides some multiplicative factors that depend on the particular lifecycle activity (from [48]).

To determine the cost of fault-tolerant software, factors are used to account for the overheads associated with the decision points and the DMs, and to account for the cost reduction in V&V caused by commonalities among variants. The commonalities include actual V&V activities (e.g., back-to-back testing) and V&V tools (e.g., test harnesses). Given the current state of the art, the values of these factors cannot be accurately estimated, but [35] provides reasonable ranges of values for the factors. The factors and their ranges from [35] are provided below.

- 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 (Section 4.2) and N self-checking programming (NSCP) (Section 4.4) when error detection is performed through comparison, and $1 < s < 1.3$ for RcB (Section 4.1) and NSCP when error detection is performed through AT (Section 7.2). This difference reflects the differences in the deciders, that is, the fact that the

deciders are specific when they decide by AT 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 gives the cost of fault-tolerant software (C_{FT}) with respect to the cost of non-fault-tolerant software (C_{NFT}):

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

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.

Table 3.3 gives the ranges for the ratio C_{FT}/C_{NFT} as well as the average values and the average values per variant. Examining this table's results provides quantitative evidence for the qualitative statement that N-variant software is less costly than N times a non-fault-tolerant software. The experimental cost results noted earlier in this section fall within the ranges noted in Table 3.3.

3.2 Programming Techniques

In this section, we describe several programming or implementation techniques used by several software fault tolerance techniques. The programming techniques covered are assertions, checkpointing, and atomic actions. Assertions can be used by any software fault tolerance technique, and by non-fault-tolerant software. Checkpointing is typically used by techniques that employ backward recovery. Atomic actions can also be used in non-fault-tolerant software, but are presented here primarily in the context of software fault tolerance in concurrent systems.

Table 3.2
Software Cost Elements for Non-Fault-Tolerant Software
(*From:* [35], © 1990, IEEE. Reprinted with permission.)

Activity	Life-Cycle Cost Breakdown [47]	Multipliers for Critical Applications [48]	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%
V&V	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 [47].

Table 3.3
Cost of Fault-Tolerant Software Versus Non-Fault-Tolerant Software
(*From:* [35], © 1990, IEEE. Reprinted with permission.)

Faults Tolerated	Fault Tolerance Method	*N*	(C_{FT}/C_{NFT}) Minimum	(C_{FT}/C_{NFT}) Maximum	(C_{FT}/C_{NFT}) Average	(C_{FT}/NC_{NFT}) Average
1	RcB	2	1.33	2.17	1.75	0.88
1	NSCP					
	AT	2	1.33	2.17	1.75	0.88
	Comparison	4	2.24	3.77	3.01	0.75
1	NVP	3	1.78	2.71	2.25	0.75
2	RcB	3	1.78	2.96	2.37	0.79
2	NSCP					
	AT	3	1.78	2.96	2.37	0.79
	Comparison	6	3.71	5.54	4.63	0.77
2	NVP	4	2.24	3.77	3.01	0.75

3.2.1 Assertions

Assertions are a fairly common means of program validation and error detection. As early as 1975, Randell [17] presented executable assertions as central to the design of fault-tolerant programs. An executable assertion is a statement that checks whether a certain condition holds among various program variables, and, if that condition does not hold, takes some action. In essence, they check the current program state to determine if it is corrupt by testing for out-of-range variable values, the relationships between variables and inputs, and known corrupted states. These assertion conditions are derived from the specification, and the assertion can be made arbitrarily stringent in its checking. Assertions may be set up to only produce a warning upon detection of a corrupt state or they may take or initiate corrective action. For example, upon the detection of a corrupt state, the assertion may halt program execution or attempt to recover from the corrupt state. What the assertion does upon firing (detecting a corrupt state) is application-dependent. Assertions can be used as part of a "reasonableness-checking AT" such as a range bounds AT (see Section 7.2.3).

Traditional assertions produce warnings when the condition being checked is not met. They do not typically attempt recovery. Recovery assertions, on the other hand, are forward recovery mechanisms that attempt to replace the current corrupt state with a correct state. As with checkpointing (discussed in the next subsection), the entire state can be replaced or only specific variables, depending on the system constraints and overhead (e.g., time and memory) involved in the saving and restoration of the variables' or state's values. Assertions can also be used to reset variables periodically (i.e., without necessarily testing for a corrupt state) in, for example, safety-critical real-time systems to limit the propagation of corrupt data values [49].

Some programming languages provide special constructs for executable assertions. However, executable assertions are essentially Boolean functions that evaluate to TRUE when the condition holds, and FALSE otherwise. Using a generic pseudocode language, we can present the simplest form of an executable assertion as

```
if not assertion then action
```

where `assertion` is a Boolean expression and `action` is a method or procedure.

The most general form of an assertion must refer to the current state and to a previous state. Primary choices for the previous state are:

- The initial state, s_0;
- An intermediate state between s_0 and the current state that was reached along the path the program execution has taken.

Mili [50] provides three reasons for which an intermediate state should be chosen over the initial state in an executable assertion:

1. *Modularity:* We can think of the assertion *a* as checking a local program segment *b* by referring to the state of the program before execution of *b* and after execution of *b*. The program segment *b* and its assertion-checking facilities then form a modular unit that is context independent—it does not depend on where it is in the program [50].
2. *Time parsimony:* Block *b* can be arbitrarily short, and the function it computes arbitrarily simple. Hence the assertion that checks it can be arbitrarily easy to compute and arbitrarily time efficient. By contrast, referring to s_0 means that, at milestone *m*, we check expected past functional properties of program *P* at *m*, whose complexity we do not choose [50].
3. *Space parsimony:* Block *b* can be arbitrarily short, and the variables it affects arbitrarily few. Hence the memory space required to save the variables modified by block *b* is arbitrarily small. By contrast, referring to s_0 means that sufficient memory space must be set aside to save all of s_0, whose size we do not choose [50].

The initial state or intermediate state, the block *b* to be checked, and the statement that saves all or part of the previous state comprise an elementary asserted block (EAB). The general form of an EAB is [50]:

```
ŝ = s;
b;  //  modifies s, but not ŝ
if not a(ŝ, s) then action;
```

In the above EAB, the assignment statement $\hat{s} = s$ means saving state s in $\hat{s}$. The expression $a(\hat{s}, s)$ is the assertion. As stated earlier, we may only want to save some variables, such as those that are going to be modified by *b* and/or those that are involved in the expression of assertion *a*.

Let's look at an example. Suppose $s \in S$ is an integer. Also, suppose that program block *b* determines the square of *s*, that is, $b = (s = s * s)$. The

following three simple assertions [50] illustrate different assertions we can use with the defined program block, *b*.

```
ŝ = s;
b;
if not (s = ŝ²) then action;

ŝ = s;
b;
if not (ŝ > 1 ⇒ s > ŝ) then action;

ŝ = s;
b;
if not (s > 0) then action;
```

In typical practice, *b* would be an intricate block of code that is difficult to analyze and $a(\hat{s}, s)$ would be a simple assertion [50].

When an error is detected in the current state, action should be taken to notify the designer (so that corrective action—fault removal—can be taken) and a procedure is invoked to perform damage assessment and take appropriate recovery action.

An assertion, *sc*, can be used to detect strict correctness (or freedom from errors) in the program. The following pseudocode sample (after [50]) illustrates the pattern for such an assertion.

```
perform_error_management
{
if not sc(ŝ = s) then {
          //  erroneous state
     produce_warning(UI_or_errorfile, detected_error);
          //  UI - User Interface
     perform_damage_assessment_and_recovery; }
}
```

3.2.2 Checkpointing

Checkpointing is used in (typically backward) error recovery, which we recall restores a previously saved state of the system when a failure is detected. Recovery points, points in time during the process execution at which the system state is saved, are established. The recovery point is discarded when the process result is accepted, and it is restored when a failure is detected. Checkpoints are one of several mechanisms used to establish these recovery points. Other mechanisms include the audit trail [51] and the recovery cache [52, 53]:

- *Checkpoint:* saves a complete copy of the state when a recovery point is established.
- *Recovery cache:* saves only the original state of the objects whose values have changed after the latest recovery point.
- *Audit trail:* records all the changes made to the process state.

In the discussion that follows, the generic term "checkpoint" will be used and will include all three mechanisms, unless otherwise stated.

The information saved by checkpoints includes the values of variables in the process, its environment, control information, register values, and so on. The information should be saved on stable storage so that even if the node fails, the saved checkpoint information will be safe. For single node, single process systems, checkpointing and recovery are simpler than in systems with multiple communicating processes on multiple nodes.

For single process checkpointing, there are different strategies for setting the checkpoints. Some strategies use randomly selected points, some maintain a specified time interval between checkpoints, and others set a checkpoint after a certain number of successful transactions have been completed. For example, [54, 55] examine the location of checkpoints based on the principle of information reduction. There is a trade-off between the frequency and amount of information checkpointed, and various performance measures (e.g., information integrity, system availability, program correctness, and expected execution time). For example, the code size between checkpoints can be a determining factor in the effectiveness and cost of a fault tolerance strategy. If the intermediate results are checked after small pieces of code have been executed, then there is lower error latency, but also a higher execution time overhead. In addition, decision points limit design diversity, since an increase in the frequency or number of decisions requires the agreement of variants at a higher level of detail. However, a large modular decomposition (and thus larger code segments between decisions or checkpoints) ensures higher variant independence and lower execution overheads. Larger code segments between checkpoints and decisions may result in a cursory acceptance test that is incapable of localizing the errors that occur. Models of the various approaches to checkpointing have been compared and their effects on system performance examined [54, 56–58].

There are generally two approaches to multiprocess backward recovery—asynchronous and synchronous checkpointing. In asynchronous checkpointing, the checkpointing by the various nodes in the system is not coordinated. However, sufficient information is maintained in the system so

that when rollback and recovery is required, the system can be rolled back to a consistent state. The cost of asynchronous checkpointing is lower than synchronous checkpointing, but the risk of unbounded rollback (the domino effect, discussed earlier in this chapter) remains. Many checkpoints for a given process may need to be saved because during rollback, a remote (in time) state can be restored. Asynchronous checkpointing is simpler than synchronous checkpointing, but can be useful *only* where expected failures are rare and there is limited communication between the system processes. State saving and restoration protocols for asynchronous checkpointing include [59–62] and others.

In synchronous checkpointing (or distributed checkpointing), establishing checkpoints is coordinated so that the set of checkpoints as a whole comprise a consistent system state. This limits the amount of rollback required, but the cost of establishing the checkpoints is higher than in asynchronous checkpointing because of the coordination required. Also, only a few checkpoints of a process need to be saved at a time. State saving and restoration protocols for synchronous checkpointing include [63–69] and others. Consistency criteria for distributed checkpoints (independent of the communication model) are investigated in [70].

Calsavara and Stroud [71] provide example C++ code for checkpointing and discuss some implementation issues. A very simple technique for implementing backward error recovery as part of a generic recovery block is to make a copy of the original object before invoking each alternate, as shown below [71] (reproduced here with permission):

```
try
{
     T oldobject = object;
     alternate(object);
     if( accept(object) )
     {
          return;
     }
}
catch (...)
{
     object = oldobject;
     continue;
}
```

An alternative implementation of backward recovery has each alternate returning a new object (shown below, [71], with permission) rather than modifying the old object in place.

```
try
{
     T newobject = alternate(object);
     if ( accept(newobject) )
     {
          object = newobject;
          return;
     }
}
catch(...)
{
     continue;
}
```

The first approach may be preferable because it only involves an initialization in the normal case, where the second approach requires both an initialization and an assignment. Both approaches make crude copies of the entire state of what could be a very large object. It is preferable to only save and restore the state that has changed.

Using C++, the designer can specify what it means to make a copy of an object. The compiler supplies default implementations of these copy operations if they are not supplied by the implementer, however, the default implementations are recursively defined in terms of their implementation for each subcomponent of the object. By overriding the assignment operator and defining the copy for classes, one has the freedom to copy objects fully or partially on demand, or to copy them using reference counting ("lazy" copy). This control may be exercised at any level of the object hierarchy. So, if each subcomponent of a large object is made up of a smaller object that makes a lazy copy of itself and only makes a full copy when it is modified, then without much effort, the copying operations used in the implementation of the recovery block algorithm above will in fact copy only the state which is modified and no more [71]. Hence, one can use these C++ mechanisms to implement a hardware recovery cache in software.

There has been much research in checkpointing over the years, for example, in protocols and performance modeling (references mentioned earlier), for different application environments (e.g., [63], mobile environments

[72]), and in object-oriented development (e.g., [71, 73, 74]). The reader is referred to the literature referenced in this section and the references therein for further details on checkpointing.

3.2.3 Atomic Actions

Atomic actions are used for error recovery, primarily in concurrent systems (and are widely used in transaction systems, which are not covered specifically in this text). Critical concurrent systems must be structured so that their complex asynchronous activities, such as those related to fault tolerance, can be achieved. One way to approach this requirement is to use atomic actions, which have been shown [75] to increase the quality and reusability of code and to reduce code complexity significantly. The activity of a group of components constitutes an atomic action if no information flows between that group and the rest of the system for the duration of the activity [76]. An *atomic action* [77–79] is an action that is:

- *Indivisible:* Either all the steps in the atomic action complete or none of them does, that is, the "all-or-nothing" property.
- *Serializable:* All computation steps that are not in the atomic action either precede or succeed all the steps in the atomic action.
- *Recoverable:* The external effects of all the steps in the atomic action either occur or not; that is, either the entire action completes or no steps are completed.

The property of atomicity guarantees that if an action successfully executes, its results and the changes it made on shared data become visible for subsequent actions. On the other hand, if a failure occurs inside of an action, the failure is detected and the action returns without changes on shared data. This enables easy damage containment and error handling, since the fault, error propagation, and error recovery all occur within a single atomic action. Therefore, the fault and associated recovery activities will not affect other system activities. If the activity of a system can be decomposed into atomic actions, fault tolerance measures can be constructed for each of the atomic actions independently [80].

The "indivisibility" property of atomic actions may seem to imply that an atomic action itself cannot have structure. However, an action can be

composed of other actions that are not necessarily primitive operations. These are called *nested atomic actions* [81]. Since a procedure (or operation or method) may invoke other procedures, which may invoke other procedures, and so on, we can naturally get a nested atomic action.

The structure of a nested action atomic cannot be visible from outside the nested atomic action. A nested atomic action consists of subactions (not visible from outside), which are seen as atomic actions to the other subactions of the same action. That is, within the nested atomic action, each subaction is an atomic action, and hence the structure of a subaction is not visible to another subaction. This enables a safe method of supporting concurrency within an action.

Although atomic actions were developed some time ago, few of the mainstream programming languages or operating systems provide direct support for atomic actions [82]. Supporting atomicity in a single process environment is straightforward. Suppose we have an action, *a*, that we wish to execute atomically. Prior to beginning the execution of *a*, checkpoint the state of the system. If no failure occurs before the completion of *a*'s execution, then the "all" part of the atomic action's "all-or-nothing" property is satisfied. If, however, a failure occurs prior to the completion of *a*'s execution, then restore the checkpointed state. This removes the effects of *a*'s partial execution and effectively satisfies the "nothing" part of the "all-or-nothing" property.

In a distributed system, supporting atomic actions is more complicated. Without proper coordination of concurrent data access, this access to shared data by different processes can cause the system state to become inconsistent. Some efforts to provide support for atomic actions in distributed systems are noted below.

- In Pact (the *p*arallel *act*ions parallel programming environment), atomic actions are used to achieve fully user-transparent fault tolerance with low run-time overhead [62].

- Wellings and Burns [83] show how atomic actions can be implemented in Ada 95, and how they can be used to implement software fault tolerance techniques.

- Avalon/C++ [84] takes advantage of inheritance to implement atomic actions in object-oriented applications.

- The Arjuna system [85] uses inheritance, in a manner similar to Avalon/C++, and object extensions to implement atomic actions in object-oriented applications.

We will examine Arjuna a bit more closely.

To implement atomic actions, the Arjuna toolkit requires the following:

- Every object used within an atomic action must be recoverable; that is, it must provide its own state preservation and recovery operations.
- Each object is also required to state explicitly within the atomic action when it is modified.
- All persistent objects are derived from a special class, LockManager, which is known to the implementation of class `AtomicAction`. `LockManager` has two pure virtual functions, `save_state` and `restore_state`, which each subclass of `LockManager` is required to define.
- To ensure that the state's save and restore operations are invoked at appropriate points during the atomic action, an object that is about to modify itself must first acquire a write lock.

There are similarities between the Arjuna mechanism and the recursive definition of default copy operations provided by the C++ compiler itself [71], described earlier.

The class declaration below [85] (boldface indicates keywords) shows the important operations provided by the Arjuna `AtomicAction` class. To create an atomic action using Arjuna, one declares an `AtomicAction` instance in the program and invokes the `Begin` operation. To create nested atomic actions, declare multiple instances of the class so nesting occurs when a `Begin` operation is invoked within the scope of another atomic action. (Code and discussion from [85] are reproduced here with permission.)

```
class AtomicAction : public StateManager
{
     // private instance variables
     RecordList List;
     …
protected:   // protected operations
     PrepareOutcome Prepare();
     void Commit();
public:
     // global class variable
     static AtomicAction *Current;
     AtomicAction();
     ~AtomicAction();

     virtual Action_Status Begin();
     virtual Action_Status End();
     virtual Action_Status Abort();
     …
     bool add(AbstractRecord*);
     AtomicAction*Parent();
     …
}
```

The following code [85] shows two atomic actions, one of which is committed (A) and the other aborted (B). Arjuna also supports nested concurrent atomic actions, implemented by the class ConcurrentAtomicAction [85].

```
AtomicAction A, B;
A.Begin();        // start of atomic action A
     B.Begin();   // start of atomic action B
     B.Abort();   // abortion of atomic action B
A.End();          // commitment of atomic action A
```

The basic structure for state-based recovery using atomic actions in Arjuna is illustrated in the pseudocode below (from [71]).

```
for (each alternate)
{
     AtomicAction A;
     A.Begin();
     try
     {
          alternate(object);
          if ( accept(object) )
          {
               A.Commit();
               return;
          }
     }
     catch(...)
     {
          A.Abort();
          continue;
     }
}
```

3.3 Dependable System Development Model and *N*-Version Software Paradigm

As stated in the introduction to this chapter, developing dependable, critical applications is not an easy task. The trend toward increasing complexity and size, distribution on heterogeneous platforms, diverse accidental and malicious origins of system failures, the consequences of failures, and the severity of those consequences combine to thwart the best human efforts at developing these applications. In this section, we describe methods to assist in the design and development of these critical, fault-tolerant software systems. In doing so, the following topics are covered: design considerations, a development model for dependable systems, and a design paradigm specific to NVP.

3.3.1 Design Considerations

This section provides a brief introduction to some of the design considerations to be examined when developing fault-tolerant software. These issues are primarily related to design diverse software fault tolerance techniques, but are also useful to consider for other types of software fault tolerance techniques.

3.3.1.1 Component Selection

One of the first major decisions to make is to determine which software functions or components to make fault tolerant. Specifications, simulation and modeling, cost-effectiveness analysis, and expert opinion from those familiar with the application and with software fault tolerance can be helpful in this determination.

3.3.1.2 Level of Application of Fault Tolerance

One of the most important early decisions impacting the system architecture is the level of fault tolerance application. There are two major questions determining the level of application of fault tolerance: (1) At what level of detail should one perform the decomposition of the system into components that will be diversified? and (2) Which layers (application software, executive, hardware) must be diversified [86]?

Level of Decomposition/Granularity

While discussing checkpointing, we touched on this question. There is a trade-off between large and small components, that is, the granularity of fault tolerance application. Component size can be a determining factor in the effectiveness and cost of a fault tolerance strategy. Small pieces of code enable lower error latency and make decision algorithms simpler and more precise. However, small components will increase the number of decision points, which limits design diversity, since an increase in the frequency or number of decisions requires the agreement of variants at a higher level of detail. A smaller modular decomposition (i.e., smaller code segments between decision or checkpoints) increases execution overhead in decision making and fault tolerance control.

Larger components favor diversity and higher variant independence. In addition, larger components result in lower execution overheads because of the lower frequency of executing the decision algorithm. However, larger component size increases the error latency because of increased synchronization delays (e.g., for NVP or NSCP) or rollback distance (e.g., for RcB). Having larger code segments between checkpoints and decisions may result in a cursory decision mechanism that is incapable of localizing the errors that occur.

Related to this decomposition/granularity decision is a lower-level design issue regarding the placement of decision and recovery points within a fault-tolerant section of code and the choice of the data upon which to perform decision making.

What Layer(s) to Diversify/Extent of Diversity

As discussed in Chapter 2, diversity can be applied at several layers of the system (e.g., the application software, executive software or operating system, and hardware) and throughout the development effort (e.g., languages, development teams and tools, and so on). Additional diversity is likely to increase reliability, but must be balanced against cost and management of the resulting diversity.

3.3.1.3 Technique Selection

Selection of which technique(s) to use is an important design consideration. This decision can be helped by input from performance analysis, simulation and modeling, cost-effectiveness analysis, design tools (e.g., [87]), and expert opinion (again, from those familiar with the application (domain) and with software fault tolerance). The information provided in Chapters 4–6 should also be useful in making this decision.

3.3.1.4 Number of Variants

Not considering any economic impact, 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 [86]. There are, of course, both cost and performance effects to consider. A larger number of variants should increase reliability (if the number of related faults does not also increase). However, a larger number of variants in a recovery block scheme will also increase the execution time and cost. Similarly, in multiversion software (such as NVP), an increase in the number of variants will result in higher development and support costs.

3.3.1.5 Design Methodology

Using a design methodology that effectively considers dependable, fault-tolerant software needs will assist in managing the complexities, realizing and handling the design and development issues particular to fault-tolerant software, and developing a dependable system. The guidance in the methodologies presented in the following sections can provide valuable assistance.

3.3.1.6 Decision Mechanism Algorithm

Selection of which DM to use is another important design consideration. This selection can be helped by input from fault-handling simulation, design tools (e.g., [87]), and expert opinion. The information provided in Chapter 7 should also be useful in selecting an appropriate DM.

3.3.1.7 Summary of Design Considerations

Little specific assistance is available to make the required decisions about design, particularly since many of them are at least partially application-dependent. However, cost and overhead information (such as that described earlier in this chapter), performance analysis (such as that described in [63]), design methodologies (e.g., those described in the next sections), and prototype design assistance tools (e.g., the Software Fault Tolerance Design Assistant (SFTDA) [87]) provide valuable guidance and input to the necessary decisions.

3.3.2 Dependable System Development Model

Given the complexity of computer-based critical software, the diversity of faults to be handled by these systems, and the consequences and severity of their failure, a systematic and structured design framework that integrates dependability concerns and requirements at the early stages of (and throughout) the development process is needed [88–90]. Software design faults are recognized as the current obstacle to successful dependable systems development [91]. Conventional development methods do not incorporate the processes and key activities required for effective development of dependable systems. To fill this need, Kaaniche, Blanquart, Laprie, and colleagues [91, 92] developed the dependability-explicit development model. The model provides guidelines emphasizing the key issues to be addressed during the main stages of dependable systems development [91, 92]. In this section, we provide an overview of the development model's key activities for the fault tolerance process and refer the reader to the sources [91, 92] for additional details and activities in other processes.

The dependability-explicit development model provides lists of key activities related to system development phases. The requirements phase begins with a detailed description of the system's intended functions and definition of the system's dependability objectives. The following list [91, 92] summarizes the key activities in the fault tolerance process for this phase.

- Description of system behavior in the presence of failures:
  - Identification of relevant dependability attributes and necessary trade-offs;
  - Failure modes and acceptable degraded operation modes;

  - Maximum tolerable duration of service interruption for each degraded operation mode;
  - Number of consecutive and simultaneous failures to be tolerated for each degraded operation mode.

The main objective of the design phase is to define an architecture that will allow the system requirements to be met. The following list [91, 92] summarizes the key fault tolerance activities and issues for this phase.

- Description of system behavior in presence of faults:
  - Fault assumptions (faults considered, faults discarded);
- System partitioning:
  - Fault tolerance structuring: fault-containment regions, error-containment regions;
  - Fault tolerance application layers;
- Fault tolerance strategies:
  - Redundancy, functional diversity, defensive programming, protection techniques and others;
- Error-handling mechanisms:
  - Error detection, error diagnosis, error recovery;
- Fault-handling mechanisms:
  - Fault diagnosis, fault passivation, reconfiguration;
- Identification of single points of failure.

The realization phase consists of implementing the system components based on the design specification. Below is a summary of the key fault tolerance process activities for the implementation or realization phase [91, 92].

- Collect the number of faults discovered during this stage:
  - Use as indicator of component dependability;
  - Use to identify system components requiring reengineering.

The integration phase consists of assembling the system components and integrating the system into its environment to make sure that the final

product meets its requirements. Following is a summary of the key fault tolerance activities for the integration phase [91, 92].

- Verification of integration of fault and error processing mechanisms:
  - Use analysis and experimentation to ensure validated fault-tolerant subsystems satisfy dependability requirements when integrated;
  - Use fault injection (multiple and near-coincident faults);
- Evaluate fault tolerance mechanisms' efficiency;
- Estimate fault tolerance mechanism coverage—fault injection experiments.

The dependability-explicit development model is provided to ensure that dependability related issues are considered at each stage of the development process. The model is generic enough to be applied to a wide range of systems and application domains and can be customized as needed. Since the key activities and guidelines of the model focus on the nature of activities to be performed and the objectives to be met, they can be applied regardless of which development methods are used.

3.3.3 Design Paradigm for *N*-Version Programming

Although the NVP technique is presented in Section 4.2, we describe in this section a design paradigm for NVP because it contains guidelines and rules that can be useful in the design of many software fault tolerance techniques. It is generally agreed that a high degree of variant independence and a low probability of failure correlation are vital to successful operation of *N*-version software (NVS). This requires attaining the lowest possible probability that the effects of similar errors in the variants will coincide at the DM. The design paradigm for NVP was developed and refined by Lyu and Avizienis [93–96] to achieve these goals. Hence, the objectives of the design paradigm, as stated in [96] are:

- To reduce the possibility of oversights, mistakes, and inconsistencies in the process of software development and testing;
- To eliminate most perceivable causes of related design faults in the independently generated versions of a program, and to identify causes of those that slip through the design process;

- To minimize the probability that two or more versions will produce similar erroneous results that coincide in time for a decision (consensus) action of the *N*-version executive (NVX).

The design paradigm for NVP is illustrated in Figure 3.7 [96]. As shown, it consists of two groups of activities. On the left side of the figure are the standard software development activities. To the right are the activities specifying the concurrent implementation of NVS. Table 3.4 summarizes the NVS design paradigm's activities and guidelines incorporated into the software development life cycle. (The table was developed by combining information found in [95] (the table structure and initial entries) and [96] (updated information on the refined paradigm). For more detail on the paradigm and a discussion of the associated issues, the reader is referred to [95, 96].

3.4 Summary

This chapter presented software fault tolerance problems and issues, programming techniques, and design and development considerations and models. The advantages of software fault tolerance are accompanied by disadvantages, issues to consider, and costs. Those common to most techniques were covered here. We covered perhaps the greatest bane of design diversity—similar errors. If these are not avoided then software fault tolerance techniques based on design diversity will not be effective. Other issues and potential problems to be considered were covered, including the CCP with FPA applications, the domino effect in backward recovery, and overhead (not just cost, but time, operation overhead, redundancy, and memory).

Then, to help in development, we described several programming methods that are used by several software fault tolerance techniques. These include assertions (that can be used by fault tolerant or non-fault-tolerant software), checkpointing (typically used in techniques employing backward recovery), and atomic actions (also used in non-fault-tolerant software, but presented here in reference to concurrent systems).

Backing out the scope, we then present methods to assist in the design and development of critical, fault-tolerant software systems. Design considerations, a development model for dependable systems, and a design paradigm specific to NVP are presented.

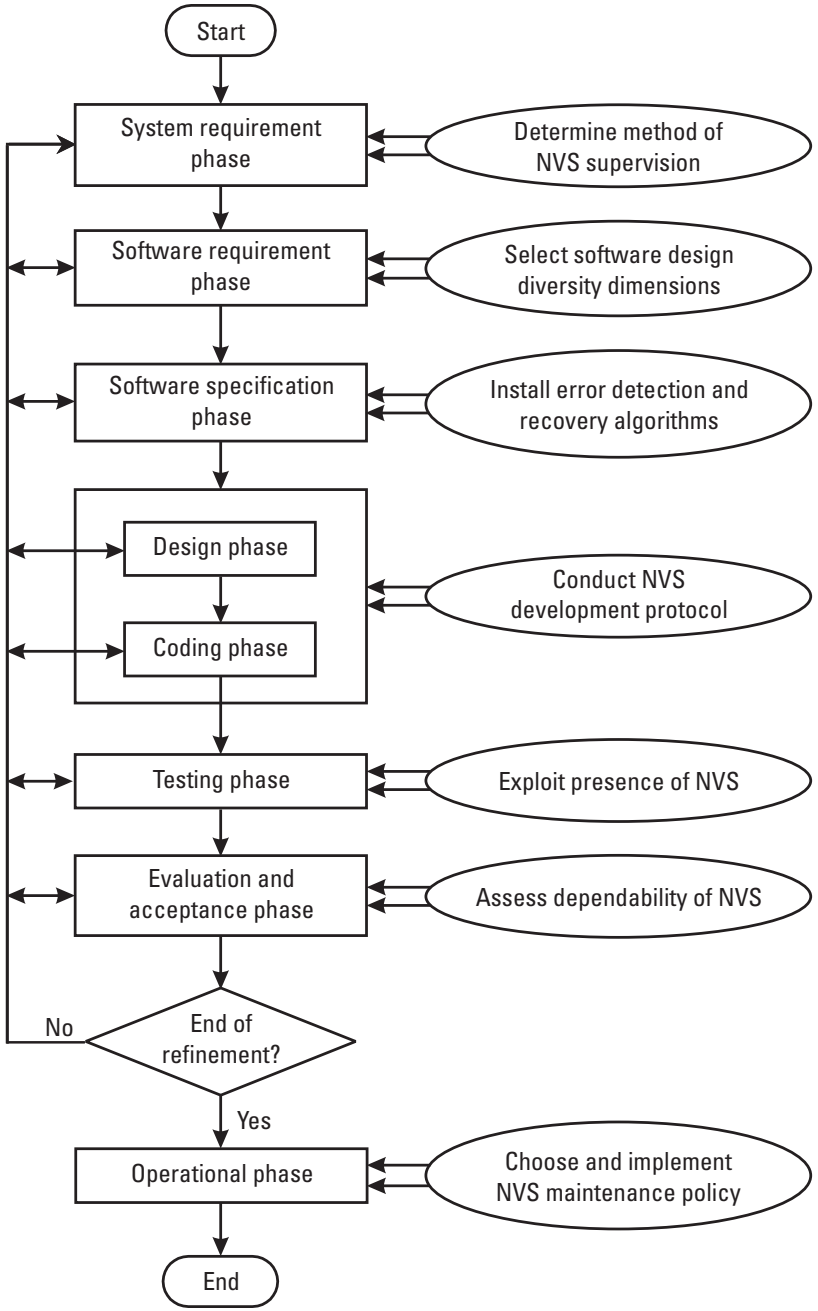

Figure 3.7 Design paradigm for *N*-version programming. (*Source:* [96], © 1995 John Wiley and Sons. Reproduced with permission.)

Table 3.4
N-Version Programming Design Paradigm Activities and Guidelines

Software Life Cycle Phase	Enforcement of Fault Tolerance	Design Guidelines and Rules
System requirement	Determine method of NVS supervision	1. Choose NVS execution method and allocate required resources 2. Develop support mechanisms and tools 3. Select hardware architecture
Software requirement	Select software design diversity dimensions	1. Assess random diversity versus required diversity 2. Evaluate required design diversity 3. Specify diversity under application constraints
Software specification	Install error detection and recovery algorithms	1. Specify the matching features needed by NVX 2. Avoid diversity-limiting factors 3. Diversify the specification
Design and coding	Conduct NVS development protocol	1. Impose a set of mandatory rules of isolation 2. Define a rigorous communication and documentation protocol 3. Form a coordinating team
Testing	Exploit presence of NVS	1. Support for verification procedures 2. Opportunities for back-to-back testing
Evaluation and acceptance	Assess the dependability of NVS	1. Define NVS acceptance criteria 2. Assess evidence of diversity 3. Make NVS dependability predictions
Operational	Choose and implement an NVS maintenance policy	1. Assure and monitor NVX functionality 2. Follow the NVP paradigm for NVS modification

This chapter has focused on issues that are fairly common across software fault tolerance techniques. In the following Chapters 4, 5, and 6, we examine individual techniques, including technique-specific issues.

References

[1] Avizienis, A., "The *N*-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, 1985, pp. 1491–1501.

[2] Chen, L., and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," *Proceedings of FTCS-8*, Toulouse, France, 1978, pp. 3–9.

[3] Avizienis, A., and L. Chen, "On the Implementation of N-Version Programming for Software Fault-Tolerance During Program Execution," *COMPSAC '77*, 1977, pp. 149–155.

[4] Avizienis, A., and J. P. J. Kelly, "Fault-Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, Vol. 17, No. 8, 1984, pp. 67–80.

[5] Eckhardt, D. E., and L. D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, 1985, pp. 1511–1517.

[6] Littlewood, B., and D. R. Miller, "Conceptual Modeling of Coincident Failures in Multiversion Software," *IEEE Transactions on Software Engineering*, Vol. 15, No. 12, 1989, pp. 1596–1614.

[7] Kelly, J. P. J., et al., "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results," *Proceedings of FTCS-18*, Tokyo, 1988, pp. 9–14.

[8] Eckhardt, D. E., et al., "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability," *IEEE Transactions on Software Engineering*, Vol. 17, No. 7, 1991, pp. 692–702.

[9] Vouk, M. A., et al., "An Empirical Evaluation of Consensus Voting and Consensus Recovery Block Reliability in the Presence of Failure Correlation," *Journal of Computer and Software Engineering*, Vol. 1, No. 4, 1993, pp. 367–388.

[10] Anderson, T., and P. A. Lee, *Fault Tolerance: Principles and Practice*, Englewood Cliffs, NJ: Prentice-Hall, 1981.

[11] Pullum, L. L., "Fault Tolerant Software Decision-Making Under the Occurrence of Multiple Correct Results," Doctoral dissertation, Southeastern Institute of Technology, 1992.

[12] Brilliant, S., J. C. Knight, and N. G. Leveson, "The Consistent Comparison Problem in *N*-Version Software," *ACM SIGSOFT Software Engineering Notes*, Vol. 12, No. 1, 1987, pp. 29–34.

[13] Brilliant, S., J. C. Knight, and N. G. Leveson, "The Consistent Comparison Problem in *N*-Version Software," *IEEE Transactions on Software Engineering*, Vol. 15, No. 11, 1989, pp. 1481–1485.

[14] Knight, J. C., and P. E. Ammann, "Issues Influencing the Use of N-Version Programming," *Information Processing '89*, 1989, pp. 217–222.

[15] Ammann, P. E., and J. C. Knight, "Data Diversity: An Approach to Software Fault Tolerance," *IEEE Transactions on Computers*, Vol. 37, No. 4, 1989, pp. 418–425.

[16] Knuth, D. E., *The Art of Computer Programming*, Reading, MA: Addison-Wesley, 1969.

[17] Randell, B., "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, 1975, pp. 220–232.

[18] Kim, K. H., "An Approach to Programmer-Transparent Coordination of Recovering Parallel Processes and Its Efficient Implementation Rules," *Proceedings IEEE Computer Society's International Conference on Parallel Processing*, 1978, pp. 58–68.

[19] Nelson, V. P., and B. D. Carroll, "Software Fault Tolerance," in V. P. Nelson and B. D. Carroll (eds.), *IEEE Tutorial on Fault Tolerant Computing*, Washington, D.C.: IEEE Computer Society Press, 1987, pp. 247–256.

[20] Randell, B., and J. Xu, "The Evolution of the Recovery Block Concept," in M. R. Lyu (ed.), *Software Fault Tolerance*, New York: John Wiley and Sons, 1995, pp. 1–21.

[21] Anderson, T., and J. Knight, "A Framework for Software Fault Tolerance in Real-Time Systems," *IEEE Trans. on Software Engineering*, Vol. SE-9, No. 5, 1983, pp. 355–364.

[22] Kelly, J. P. J., T. I. McVittie, and W. I. Yamamoto, "Implementing Design Diversity to Achieve Fault Tolerance," *IEEE Software*, July 1991, pp. 61–71.

[23] Levi, S.-T., and A. K. Agrawala, *Fault-Tolerant System Design*, New York: McGraw-Hill, 1994.

[24] Kim, K. H., "Approaches to Mechanization of the Conversation Scheme Based on Monitors," *IEEE Trans. on Software Engineering*, Vol. SE-8, No. 5, 1993, pp. 189–197.

[25] Kim, K. H., "Distributed Execution of Recovery Blocks: Approach to Uniform Treatment of Hardware and Software Faults," *Proceedings IEEE 4th International Conference on Distributed Computing Systems*, 1984, pp. 526–532.

[26] Kim, K. H., "Programmer-Transparent Coordination of Recovering Concurrent Processes: Philosophy & Rules," *IEEE Transactions on Software Engineering*, Vol. 14, No. 6, 1988, pp. 810–817.

[27] Merlin, P. M., and B. Randell, "State Restoration in Distributed Systems," *Proceedings of FTCS-8*, Toulouse, France, 1978, pp. 129–134.

[28] Laprie, J. -C., et al., "Architectural Issues in Software Fault Tolerance," in M. R. Lyu (ed.), *Software Fault Tolerance*, New York: John Wiley & Sons, 1995, pp. 47–80.

[29] Xu, J., A. Bondavalli, and F. Di Giandomenico, "Dynamic Adjustment of Dependability and Efficiency in Fault-Tolerant Software," in B. Randell, et al. (eds.), *Predictably Dependable Computing Systems*, New York: Springer-Verlag, 1995, pp. 155–172.

[30] Halton, L., "N-Version Design Versus One Good Design," *IEEE Software*, Nov./Dec. 1997, pp. 71–76.

[31] Bishop, P. G., et al., "PODS—A Project on Diverse Software," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, 1986, pp. 929–940.

[32] Hagelin, G., "Ericsson Safety System for Railway Control," in U. Voges (ed.), *Software Diversity in Computerized Control Systems*, Vienna, Austria: Springer-Verlag, 1988, pp. 11–21.

[33] Voges, U., "Software Diversity," *Reliability Engineering and System Safety*, Vol. 43, 1994.

[34] Panzl, D. J., "A Method for Evaluating Software Development Techniques," *The Journal of Systems Software*, Vol. 2, 1981, pp. 133–137.

[35] Laprie, J. -C., et al., "Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures," *IEEE Computer*, Vol. 23, No. 7, 1990, pp. 39–51.

[36] Anderson, T., et al., "Software Fault Tolerance: An Evaluation," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, 1985, pp. 1502–1510.

[37] Avizienis, A., et al., "DEDIX 87—A Supervisory System for Design Diversity Experiments at UCLA," in U. Voges (ed.), *Software Diversity in Computerized Control Systems*, Vienna, Austria: Springer-Verlag, 1988, pp. 127–168.

[38] Bhargava, B., and C. Hua, "Cost Analysis of Recovery Block Scheme and Its Implementation Issues," *International Journal of Computer and Information Sciences*, Vol. 10, No. 6, 1981, pp. 359–382.

[39] McAllister, D. F., "Some Observations on Costs and Reliability in Software Fault-Tolerant Techniques," *Proceedings TIMS-ORSA Conference*, Boston, MA, 1985.

[40] Saglietti, F., and W. Ehrenberger, "Software Diversity—Some Considerations about Benefits and Its Limitations," *Proceedings IFAC SAFECOMP '86*, Sarlet, France, 1986, pp. 27–34.

[41] Vouk, M. A., "Back-to-Back Testing," *Journal of Information and Software Technology*, Vol. 32, No. 1, 1990, pp. 34–45.

[42] McAllister, D. F., and R. K. Scott, "Cost Models for Fault-Tolerant Software," *Journal of Information and Software Technology*, Vol. 33, No. 8, 1991, pp. 594–603.

[43] Lyu, M. R. (ed.), *Software Fault Tolerance*, New York: John Wiley & Sons, 1995.

[44] Betts, A. E., and D. Wellbourne, "Software Safety Assessment and the Sizewell B Applications," *International Conference on Electrical and Control Aspects of the Sizewell B PWR*, Churchill College, Cambridge, 1992.

[45] Ward, N. J., "Rigorous Retrospective Static Analysis of the Sizewell B Primary Protection System Software," *Proceedings IFAC SAFECOMP '93*, Poznan-Kiekrz, Poland, 1993.

[46] Kanoun, K., "Cost of Software Design Diversity: An Empirical Evaluation," LAAS Report No. 9163, Toulouse, France: LAAS, 1999.

[47] Ramamoorthy, C. V., et al., "Software Engineering: Problems and Perspectives," *Computer*, Vol. 17, No. 10, 1984, pp. 191–209.

[48] Boehm, B. W., *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice-Hall, 1981.

[49] Parnas, D. L., A. J. Van Schouwen, and A. Po Kwan, "Evaluation of Safety-Critical Software," *Communications of the ACM*, June 1990, pp. 636–648.

[50] Mili, A., *An Introduction to Program Fault Tolerance: A Structured Programming Approach*, New York: Prentice-Hall, 1990.

[51] Bjork, L. A., "Generalized Audit Trail Requirements and Concepts for Data Base Applications," *IBM Systems Journal*, Vol. 14, No. 3, 1975, pp. 229–245.

[52] Horning, J., et al., "A Program Structure for Error Detection and Recovery," in *Lecture Notes in Computer Science*, Vol. 16, New York: Springer-Verlag, 1974, pp. 171–187.

[53] Lee, P. A., N. Ghani, and K. Heron, "A Recovery Cache for the PDP-11," *IEEE Transactions on Computers*, June 1980, pp. 546–549.

[54] Saglietti, F., "Location of Checkpoints by Considering Information Reduction," in M. Kersken and F. Saglietti (eds.), *Software Fault Tolerance: Achievement and Assessment Strategies,* New York: Springer-Verlag, 1992, pp. 225–236.

[55] Saglietti, F., "Location of Checkpoints in Fault-Tolerant Software," *IEEE*, 1990, pp. 270–277.

[56] Nicola, V. F., and J. M. Spanje, "Comparative Analysis of Different Models of Checkpointing and Recovery," *IEEE Transactions on Software Engineering*, Vol. 16, No. 8, 1990, pp. 807–821.

[57] Nicola, V. F., "Checkpointing and the Modeling of Program Execution Time," in M. R. Lyu (ed.), *Software Fault Tolerance*, New York: John Wiley & Sons, 1995, pp. 167–188.

[58] Kulkarni, V. G., V. F. Nicola, and K. S. Trivedi, "Effects of Checkpointing and Queuing on Program Performance," *Communications on Statistics—Stochastic Models*, Vol. 6, No. 4, 1990, pp. 615–648.

[59] Wood, W. G., "A Decentralized Recovery Protocol," *Proceedings of FTCS-11*, Portland, OR, 1981, pp. 159–164.

[60] Juang, T. T. Y., and S. Venkatesan, "Efficient Algorithm for Crash Recovery in Distributed Systems," *10th Conference on Foundations of Software Technology and Theoretical Computer Science (LNCS)*, 1990, pp. 349–361.

[61] Juang, T. T. Y., and S. Venkatesan, "Crash Recovery with Little Overhead," *11th International Conference on Distributed Computing Systems*, 1991, pp. 454–461.

[62] Maier, J., "Fault-Tolerant Parallel Programming with Atomic Actions," in D. Pradhan and D. Avresky (eds.), *Fault-Tolerant Parallel and Distributed Systems*, Los Alamitos, CA: IEEE Computer Society Press, 1995, pp. 210–219.

[63] Tai, A. T., J. F. Meyer, and A. Avizienis, *Software Performability: From Concepts to Applications*, Norwell, MA: Kluwer Academic Publishers, 1996.

[64] Chandy, K. M., and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, Vol. 3, No. 1, 1985, pp. 63–75.

[65] Koo, R., and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, 1987, pp. 23–31.

[66] Cristian, F., and F. Jahanian, "A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations," *Proceedings of Reliable Distributed Software and Database Systems*, Pisa, Italy, 1991, pp. 12–20.

[67] Tong, Z., R. Y. Kain, and W. T. Tsai, "Rollback Recovery in Distributed Systems Using Loosely Synchronized Clocks," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 2, 1992, pp. 246–251.

[68] Venkatesh, K., T. Radhakrishnan, and H. F. Li, "Optimal Checkpointing and Local Recording for Domino-Free Rollback Recovery," *Proceedings of FTCS-11*, Portland, OR, 1981, pp. 159–164.

[69] Shin, K. G., and Y. -H. Lee, "Evaluation of Error Recovery Blocks used for Cooperating Processes," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 11, 1984, pp. 692–700.

[70] Helary, J.-M., R. H. B. Netzer, and M. Raynal, "Consistency Issues in Distributed Checkpoints," *IEEE Transactions on Software Engineering*, Vol. 25, No. 2, 1999, pp. 274–281.

[71] Calsavara, C. M. F. R., and R. J. Stroud, "Forward and Backward Error Recovery in C++," University of Newcastle upon Tyne, Technical Report No. 417, 1993.

[72] Ssu, K. -F., et al., "Adaptive Checkpointing with Storage Management for Mobile Environments," *IEEE Transactions on Reliability*, Vol. 48, No. 4, 1999, pp. 315–323.

[73] Kasbekar, M., C. Narayanan, and C. R. Das, "Selective Checkpointing and Rollbacks in Multi-Threaded Object-Oriented Environment," *IEEE Transactions on Reliability*, Vol. 48, No. 4, 1999, pp. 325–339.

[74] Garcia, J. C. R., et al., *Optimized Object State Checkpointing Using Compile-Time Reflection*, LAAS Report 98173, 1998.

[75] Rafnel, B. A., "A Transaction Approach to Error Handling," *Hewlett-Packard Journal*, June 1993, pp. 71–77.

[76] Lee, P. A., and T. Anderson, *Fault Tolerance: Principles and Practice*, 2nd ed., New York: Springer-Verlag, 1990.

[77] Lomet, D. B., "Process Structuring, Synchronization and Recovery Using Atomic Actions," *ACM SIGPLAN Notices*, Vol. 12, No. 3, 1977, pp. 128–137.

[78] Liskov, B., and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust Distributed Programs," *ACM TOPLAS*, Vol. 5, No. 3, 1983, pp. 381–404.

[79] Reed, D. P., "Implementing Atomic Actions on Decentralized Data," *ACM TOCS*, Vol. 1, No. 1, 1975, pp. 3–23.

[80] Campbell, R. H., and B. Randell, "Error Recovery in Asynchronous Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 8, 1986, pp. 811–826.

[81] Moss, J. E. B., *An Introduction to Nested Transactions*, Technical Report 86-41, Amherst, MA: University of Amherst Massachusetts, 1986.

[82] Burns, A., and A. J. Wellings, *Real-Time Systems and Their Programming Languages*, Reading, MA: Addison-Wesley, 1990.

[83] Wellings, A. J., and A. Burns, "Implementing Atomic Actions in Ada 95," *IEEE Transactions on Software Engineering*, Vol. 23, No. 2, 1997, pp. 107–123.

[84] Detlefs, D., M. P. Herlihy, and J. M. Wing, "Inheritance of Synchronization and Recovery Properties in Avalon/C++," *IEEE Computer*, Vol. 21, No. 2, 1988, pp. 57–69.

[85] Shrivastava, S. K., G. D. Dixon, and G. D. Parrington, "An Overview of the Arjuna Distributed Programming System," *IEEE Software*, Vol. 8, No. 1, 1991, pp. 66–73.

[86] Laprie, J. -C., et al., "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures," in B. Randell et al. (eds.), *Predictably Dependable Computing Systems*, New York: Springer, 1995, pp. 103–122.

[87] Pullum, L. L., *Software Fault Tolerance Design Assistant (SWFTDA) User's Manual*, Quality Research Associates Technical Report, QRA-SWFTDA-SUM, 1997.

[88] Avizienis, A., "Building Dependable Systems: How to Keep Up with Complexity," *Proceedings of FTCS-25—Special Issue*, Pasadena, CA, 1995, pp. 4–14.

[89] Laprie, J. -C., "Dependability of Computer Systems: from Concepts to Limits," *1998 IFIP International Workshop on Dependable Computing and Its Applications (DCIA98)*, Johannesburg, South Africa, 1998, pp. 108–126.

[90] Laprie, J. -C., "Software-Based Critical Systems," *Proceedings of the 15th International Conference on Computer Safety, Reliability and Security (SAFECOMP'96)*, Vienna, Austria, 1996, pp. 157–170.

[91] Kaaniche, M., J. -P. Blanquart, and J. -C. Laprie, *A Dependability-Explicit Development Model*, LAAS Report 98341, 1998.

[92] Kaaniche, M., J. -C. Laprie and J. -P. Blanquart, "A Dependability-Explicit Model for the Development of Computing Systems," *19th International Conference on Computer Safety, Reliability and Security (SAFECOMP-2000)*, Rotterdam, NL, 2000, pp. 109–116.

[93] Lyu, M. R., "A Design Paradigm for Multi-Version Software," Ph. D. dissertation, University of California in Los Angeles, 1988.

[94] Lyu, M. R., and A. Avizienis, "Assuring Design Diversity in N-Version Software: A Design Paradigm for N-Version Programming," in J. F. Meyer and R. D. Schlichting (eds.), *Dependable Computing for Critical Applications 2*, New York: Springer-Verlag, 1992, pp. 197–218.

[95] Lyu, M. R., and Y. -T. He, "Improving the N-Version Programming Process Through the Evolution of a Design Paradigm," *IEEE Transactions on Reliability*, Vol. 42, No. 2, 1993, pp. 179–189.

[96] Avizienis, A., "The Methodology of N-Version Programming," in M. R. Lyu (ed.), *Software Fault Tolerance*, New York: John Wiley and Sons, 1995, pp. 23–46.