Chapter

14

Fault-Tolerant Software Reliability Engineering

D. F. McAllister and M. A. Vouk
North Carolina State University

14.1 Introduction

Software development processes and methods have been studied for decades. Despite that, we still do not have reliable tools to guarantee that complicated software systems are fault-free. In fact, it may never happen that we will be able to guarantee error-free software. The reason is that the two basic ways of showing that software is correct, proof of program correctness and exhaustive testing, may never be practical for use with very complex software-based systems, although reuse of reliable software building blocks (objects) may go a long way toward achieving that goal. Techniques for proving software correct tend to work only for relatively small and simple synchronous systems, while testing methods, although increasingly more sophisticated, do not guarantee production of error-free code because exhaustive testing is not practical in most cases. Therefore, it is reasonable to investigate techniques that permit software-based systems to operate reliably and safely even when (potential) faults are present.

A way of handling unknown and unpredictable software (and hardware) failures (faults) is through fault tolerance. Over the last two decades, there has been a considerable amount of research, as well as practical software engineering, in this area. In this chapter, we introduce some elementary principles that underlie construction of fault-tolerant software based on the software diversity principle, that is, provision of fault tolerance through functional redundancy. We leave the details of advanced analyses and the details of practical implementations to other texts. The reader who wishes to implement fault tolerance in a software-based system in practice is strongly advised to

consult additional texts such as [Eckh85, Voge87a, Litt89, Lapr90a, Mili90, Eckh91, Lapr92, Siew92, Lyu95a] (and references therein) which provide a *detailed* discussion of the more recent advances and problems in practical software fault tolerance, hardware–software interactions, and fault tolerance for distributed systems.

In this chapter, we first provide some background information, including an overview of major industrial and academic efforts related to fault-tolerant software. We then present the principles and terminology, and give a general overview of the more common techniques for tolerating software faults. This is followed by a discussion of more advanced techniques, and then by some techniques which can be used in modeling the behavior of fault-tolerant software. Finally, we discuss issues such as independence of failures, and issues related to development and cost of fault-tolerant software.

14.2 Present Status

Fault-tolerant software has been considered for use in a number of critical application areas. For example, in nuclear power plants [Gmei79, Bish86], in railway systems [Hage87], and in aerospace systems [Mart82, Will83, Bric84, IEEE84, Spec84, Madd84, Kapl85, Troy85, Aviz87, Aviz88, Davi93]. Overviews of the use of software diversity in computerized control systems can be found in [Vog87a, Lyu95a].

A number of systematic experimental studies of fault-tolerant software issues have been conducted over the last 20 years by both academia and industry. For example, experiments related to the use of fault-tolerant software were reported for nuclear industry applications [Bish86, Bish88, Voge87c], aerospace applications [Knig86, Shim88, Kell88, Aviz88, Bril90, Vouk90a, Eckh91, Lyu92a, Huda93], as well as in other areas [Ande85, Lee93].

It would appear that the first industrial use of fault-tolerant software, based on the software diversity principle, occurred in a railway system [Voge87b, Ster78]. A number of organizations have used the approach to either help develop, verify, or actually implement an operational railway application for deployment in Sweden, Denmark, Finland, Switzerland, Turkey and Bulgaria [Hage87], Italy [Frul84], Singapore [Davi84], and the United States [Turn87].

Use of fault-tolerant software in aerospace applications has received a lot of attention over the years. It has been considered and implemented in both military (e.g., [Mart82, Turn87]) and civilian (e.g., [Hill83, Wrig86, Will83, Youn86, Swee95]) aircraft, and in the U.S. space shuttle [Madd84, Spec84]. For example, the slat/flap control system for the civilian Airbus A310 airliner consists of two functionally identical computers with diverse hardware and software [Wrig86, Trav87]. Also, fault tolerance is an essential part of the experimental flight technology

such as that found in forward-swept wing, aerodynamically unstable but very agile, aircraft [Kapl85, Davi93]; and fly-by-wire military aircraft invariably incorporate fault tolerance, although not necessarily fault-tolerant software. Another example is the NASA space shuttle. The shuttle carries a configuration of four identical flight computers, each loaded with the same software, to combat hardware failures, and a fifth computer developed by a different manufacturer and running dissimilar (but in part functionally equivalent) software, which is executed only if the software in the other four processors cannot reach consensus during critical phases of the flight [Spec84, Madd84]. Software diversity was a salient issue in all developments mentioned above. We discuss this concept in more detail below.

Practical experiences with fault-tolerant software appear to be mixed but, in our opinion, they are more positive than negative, although a number of issues remain unresolved. *The general consensus appears to be that fault-tolerant software has the capability of increasing the reliability of a computer-based system.* Open and controversial issues include items such as how much fault-tolerant software actually increases system reliability in practice [Butl93], whether fault-tolerant software should be used in critical systems at all [IEEE94], and which fault-tolerance mechanism to use and how cost-effective it is [Lapr90a, Voge87b].

The primary reason for these doubts about redundancy-based software methods is the potential for common-cause faults and correlated coincident failures among the software elements that provide the redundancy. Unlike hardware failures, software faults are for the most part the result of software specification and design errors, and thus simple replication of software components does not provide reasonable protection. This dictates the need to strive for designs and development methods that encourage the use of diverse algorithms in redundant components and to minimize the potential for common-cause faults. Many experiments with fault-tolerant software have reported failure correlation among software versions used to provide redundancy (e.g., [Scot84a, Scot84b, Vouk85, Knig86, Vouk86a, Bish91, Eckh91, Gers91]), but the origins and the extent of that correlation in practical systems is still not well understood. It is conjectured, however, that failure correlation is a strong function of, among other things, software process and methods employed in version development [Lyu92b, Lyu93a]. Improving this development process thus can effectively increase overall system reliability.

14.3 Principles and Terminology

A principal way of introducing fault-tolerant software into an application is to provide a method to dynamically determine if the software is producing correct output, that is, a self-checking or *oracle* capability [Yau75]. This is often accomplished through a combination of different,

but *functionally equivalent,* software *alternates, components, versions,* or *variants* and run-time comparisons among their results. However, other techniques, ranging from mathematical consistency checking to coding, are also useful [Ande81, Lin83, Mili90, Lapr90a, Lapr92a, Adam93] as are methods which use data diversity [Amma87, Chri94] (see Probs. 14.1 and 14.2).

When software execution encounters a software fault or defect, very often the system will make a transition into an erroneous (internal) state, that is, an unexpected internal result will be created. If this erroneous result propagates and is eventually observed by the user of the system, we say that we have observed a system failure [Aviz84]. Once an erroneous state has been identified, error *recovery* can be initiated. It may involve *backward recovery,* that is, system states are saved at predetermined *recovery points,* and on detection of an erroneous state the system is rolled back or restored to a previously saved recovery point and then restarted from that state. An alternative approach is *forward recovery.* Forward recovery may be implemented as a transition into a new system state in which the software can operate (often in a degraded mode), or by *error compensation* based on an algorithm that uses redundancy built into the system to derive the correct answer. A special case of the latter is permanent *fault masking* where compensation takes place regardless of whether an error is detected or not (e.g., certain forms of voting) [Lapr90a]. Combinations of forward and backward recovery are also used.

An important factor in determining how to detect and handle errors is whether the erroneous state results from algorithmic or implementational *memory,* or not. An algorithm or a program is said to be "memoryless" if it uses only the data received or generated after the last time it has delivered a result. An example of a memoryless algorithm is process monitoring where tasks begin based on current sensor data and do not use data from previous processing [Lapr90a].

Important criteria for judging suitability of a fault-tolerance scheme are the processing overhead required to implement fault tolerance, nature of the error detection mechanism, whether correctness of the result is determined in an absolute or relative manner (e.g., with respect to specifications, or another version), whether the scheme is sequential or parallel, whether error control involves suspension of the service or not, whether results are presented within time constraints, and how many errors can be tolerated and at what cost [Lapr90a].

14.3.1 Result verification

14.3.1.1 Acceptance testing. The most basic approach to self-checking is through an (internal) *acceptance test.* An acceptance test is a pro-

grammer-provided, program-specific, error-detection mechanism, that provides a check on the results of program execution. An acceptance test might only consist of bounds or simple tests to determine acceptability. For instance, it is much easier to develop software that determines if a list is sorted than it is to develop software that performs the sorting. However, in general an acceptance test can be complex and as costly to develop as the full problem solution. An important characteristic of an acceptance test is that it uses only the data that are also available to the program at run time.

An interesting example is the following implicit specification of the square root function [Fair85], SQRT, which can serve as an acceptance test:

$$\text{for } (0 \le x \le y) \qquad (\text{ABS}((\text{SQRT}(x)*\text{SQRT}(x)) - x) < E) \qquad (14.1)$$

where E is the permissible error range, and x and y are real numbers. If E is computed for the actual code generated for the SQRT program, and machine specifications are known to the program, then Eq. (14.1) can serve as a full self-checking (acceptance) test. If E is known only to the programmer (or tester), and at run time the program does not have access to machine specifications or the information about the allowable error propagation within the program, then the test is either partial or cannot be executed by the program at general run time at all.

Acceptance tests that are specifically tailored to an algorithm are sometimes called *algorithmic* [Abra87, Huda93]. For example, provision of checksums for rows and columns of a matrix can facilitate detection of problems and recovery [Huan84]. Similarly, use of redundant links in linked lists can help combat partial loss of list items [Tayl80].

14.3.1.2 External consistency. An *external consistency check* is an extended error-detection mechanism. It may be used to judge the correctness of the results of program execution, but only with some outside intervention. It is a way of providing an oracle for off-line and development testing of the software, as well as for run-time *exception handling.* In situations where the exact answer is difficult to compute beforehand, it may be the most cost-effective way of validating components of a fault-tolerant system, or checking on run-time results.

A consistency check may use *all* information, including information that may not be available to the program in operation, but which may be available to an outside agent. Examples are watchdog processes that monitor the execution of software-based systems and use information that may not be available to software, such as timing, to detect and resolve problems [Upad86, Huda93]. Another example is periodic

entry (manual or automatic) of location coordinates to a navigation system. Yet another example is an exception signal raised by the computer hardware or operating system when floating-point or integer overflow and divide-by-zero are detected.

The interrupt signal, or exception, that abnormal events occurring in a computer system generate represents a particularly useful and prevalent resource for external consistency checking. Often, these signals can be detected and trapped in software and the exceptions can be handled to provide a form of failure tolerance. For example, many modern programming languages* allow trapping of floating-point exceptions, divide-by-zero exceptions, or IEEE arithmetic signals (such as NANs†) [IEEE85, IEEE87a], and invocation of appropriate exception handlers that provide an alternative computation or other action, and thus shield the users from this type of run-time error. A good example of applied exception handling in FORTRAN is found in [Hull94].

Consistency checking may include comparisons against exact results, but more often it involves use of knowledge about the exact nature of the input data and conditions, combined with the knowledge of the transformation (relationship) between the input data and the output data. The consistency relationship must be sufficient to assert correctness.

For example, suppose that navigational software of an aircraft samples accelerometer readings, and from that computes its estimate of the aircraft acceleration [Eckh91]. Let an acceleration vector estimate, $\hat{\mathbf{x}}$, be given by the least squares approximation

$$\hat{\mathbf{x}} = [\mathbf{C}^T\mathbf{C}]^{-1}\mathbf{C}^T\mathbf{y} \tag{14.2}$$

The matrix $\mathbf{C}$ is the transformation matrix from the instrument frame to the navigation frame of reference, $\mathbf{C}^T$ is its transpose, -1 denotes matrix inverse, and the sensor measurements are related to the true acceleration vector $\mathbf{x}$ by

$$\mathbf{y} = \mathbf{C}\mathbf{x} + \tilde{\mathbf{y}} \tag{14.3}$$

where $\tilde{\mathbf{y}}$ is the sensor inaccuracy caused by noise, misalignment, and quantization. Then,

$$\mathbf{C}^T\mathbf{C}(\hat{\mathbf{x}} - \mathbf{x}) = \mathbf{C}^T\tilde{\mathbf{y}} \tag{14.4}$$

is a criterion to assert correctness for acceleration estimates. Note that $\mathbf{x}$ and $\tilde{\mathbf{y}}$ are not normally available to the navigation software. How-

* For example, UNIX signals can be trapped in C (e.g., see standard "man" pages for "signal(3)" system calls).

† NAN is an acronym for not-a-number, used to describe arithmetic exception events that do not result in numbers.

ever, if we supply all information, including that pertaining to the environment, we can control $\mathbf{x}$ and $\tilde{\mathbf{y}}$ and detect problems with algorithms without having advance knowledge of the correct answer. This can provide oracle capabilities during the off-line testing when the environment is completely under control. Of course, such a test cannot be employed during operational use of software unless an accurate environment status is provided independently.

On the contrary, hardware and operating system *exceptions,* such as overflow and underflow, *can and should* be handled at run time, and appropriate exception handling algorithms should be part of any software system that strives to provide a measure of failure tolerance.

14.3.1.3 Automatic verification of numerical results. A rather special set of techniques for dynamic detection and control of numerical failures* is *automatic verification of numerical precision.* This type of verification treats errors resulting from algorithmic micromemory, that is, error propagation within a numerical algorithm, and possibly numerical algorithm instability. The essence of the problem is that verification of a numerical algorithm does not guarantee in any way its numerical correctness unless its numerical properties are explicitly verified.

Floating-point arithmetic is a very fast and very commonly used approach to scientific and engineering calculations. As a rule, *individual* floating-point operations made available in modern computers are maximally accurate, and yet it is quite possible for the reliability of numerical software to be abysmally poor because a series of consecutive floating-point operations delivers completely wrong results due to rounding errors and because large numbers swamp small ones. To illustrate the issue we consider the following example from [Kuli93, Adam93].

Let $\mathbf{x}$ and $\mathbf{y}$ be two vectors with six components, $\mathbf{x} = (10^{20}, 1223, 10^{24}, 10^{18}, 3, -10^{21})$, and $\mathbf{y} = (10^{30}, 2, -10^{26}, 10^{22}, 2111, 10^{19})$. The scalar product of these two vectors is defined as

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{6} x_i \cdot y_i \tag{14.5}$$

The correct answer is 8779. However, implementation of this expression on practically every platform available today will return zero unless special precautions are taken. The reason is the rounding coupled with

* An example of a numerical software-associated life-critical failure is the tragic death of a number of American service personnel during the 1991 Gulf War due to a failure of a Patriot antimissile missile to destroy an incoming enemy surface-to-surface missile because of a numerical error [GAO92].

the large difference in the order of magnitude of the summands. This happens despite the fact that each individual number can be quite comfortably represented within the floating-point format of all platforms.

It is possible to construct more subtle and more complex examples which show that simple numerical algorithms, such as Newton's method, can become very unstable if ordinary floating-point arithmetic is used without explicit error propagation control [Kuli93]. The arithmetic error propagation problem can become very acute in applications such as simulation and mathematical modeling, and it is exacerbated by modern high-speed computers.

A solution proposed in [Kuli81, Klat91, Klat92, Adam93, Kuli93] relies on the computation of the optimal dot product using fixed-point accumulation to *guarantee* maximal computational accuracy [Kuli81] on interval arithmetic (which implies certain rounding rules) to compute accurate upper and lower bounds on the result and on automatic differentiation methods [Grie92, Adam93]. If the computed result cannot be verified to be correct (for example, the computed bounds are too large), the user can be given the option of providing alternatives to the algorithms and methods, or the option of changing to higher-precision arithmetic followed by reverification of results and a decision regarding the acceptability of the results. This approach dynamically tolerates and controls this type of failure. Other solutions are available. Anyone involved with design of critical systems that use numerical software is strongly advised to consult relevant literature (e.g., [Cart83, Grie92, Adam93]).

14.3.2 Redundancy

The technique of using redundant software modules as a protection against residual software faults was inherited from hardware. Hardware faults are usually random (e.g., due to component aging). Therefore using identical backup units, or redundant spare units, with automatic replacement of failed components at run time, is a sensible approach (e.g., [Triv82, Nels87, Siew92]). However, replication of a software version to provide backup redundancy has limited effectiveness since software faults are almost exclusively design- and implementation-related and therefore would also be replicated. The net effect would be that excitation of a replicated fault would result in a simultaneous failure of all versions and there would be no fault tolerance. This does not include timing or transient faults, which often occur because of complex hardware/software/operating system interaction. Such failures (called *Heisenbugs* in [Gray90]) can rarely be duplicated or diagnosed. A common solution is to reexecute the software in the hope that the transient disturbance is over.

A solution proposed specifically for software was to have independent manufacturers produce functionally equivalent software components* [Rand75, Aviz77]. It was conjectured that different manufacturers would use different algorithms and implementation details, and that the residual faults from one independent software manufacturer would be different from those made by another; therefore when one version failed, its backup or spare would *not* fail for the same input data and conditions and would become the primary module, or at least could force a signal of the disagreement. The goal is to make the modules as diverse as possible. The overall philosophy is to enhance the probability that the modules fail on *disjoint* subsets of the input space, and thus have at any time at least one correctly functioning software component.

Specifications used in this process may themselves be diverse as long as final functional equivalency of the products is preserved. The specification indicates certain critical outputs which must be presented to an adjudication program to determine if the system is operating correctly. Each developer creates a software module or version which implements the specification and provides the outputs indicated by the specification.

Redundancy requires the ability to judge acceptability of the outputs of several modules either by direct evaluation or by comparison. The algorithm that compares or evaluates outputs is called an *adjudicator.* Such a program can be very simple or very complex depending on the application, and therefore can also be a source of errors. An adjudication program may use the outputs from redundant versions to determine which, if any, are correct or safe to pass on to the next phase of the software. The decision may be based on several different algorithms. There are several adjudication techniques which have been proposed. This includes voting, selection of the median value, and acceptance testing, as well as more complex decision making. In all situations, of course, problems arise if failures are coincidental, correlated, or similar.

14.3.3 Failures and faults

We use the terms *coincident, correlated,* and *dependent* failures (faults) as follows. When two or more functionally equivalent software components fail on the *same* input case, we say that a *coincident* failure has occurred. Failure of k components raises a k-fold coincident failure. When two or more versions give the same incorrect response (to a given tolerance) we say that an *identical-and-wrong* (IAW) answer was

* We use the terms *component(s), alternate(s), version(s), variant(s),* and *module(s)* interchangeably in this chapter.

obtained. If the measured probability of the coincident failures is significantly different from what would be expected by random chance, usually based on the measured failure probabilities of the participating components, then we say that the observed coincident failures are *correlated.* Note that two events can be correlated because they *directly depend* on each other, or because they both depend on some other, but same, event(s) (*indirect dependence*), or both.

Let $P\{\ \}$ denote probability. Then

$$P\{\text{ version}(i)\text{ fails } \mid \text{ version}(j)\text{ fails }\} \neq P\{\text{ version}(i)\text{ fails }\} \quad (14.6)$$

means that the conditional probability that version i fails given that version j has failed is different from the probability that version i, considered on its own, fails on the same inputs. If this relationship is true, we do not have failure independence [Triv82].

We shall say that several components contain the *same* or *similar fault* or *common-cause fault* if the fault's nature, and the variables and function(s) it affects, are the same for all the involved components. The result (answer) of execution of common-cause faults may be identical (IAW to within tolerance), or may be different. It is also possible that different faults result in a coincident failure by chance, giving either different answers or IAW answers.

Possible failure events are illustrated in Fig. 14.1. The Venn diagram shown in the figure represents the overlaps in the failure space of three functionally equivalent software units (or a 3-*tuple* of variants). Unshaded areas are regions where one of the three components (programs p_1, p_2, p_3) fails for an input. Lightly shaded areas show the regions where two out of three components fail coincidentally, while the darkly shaded area in the middle is the region where all three

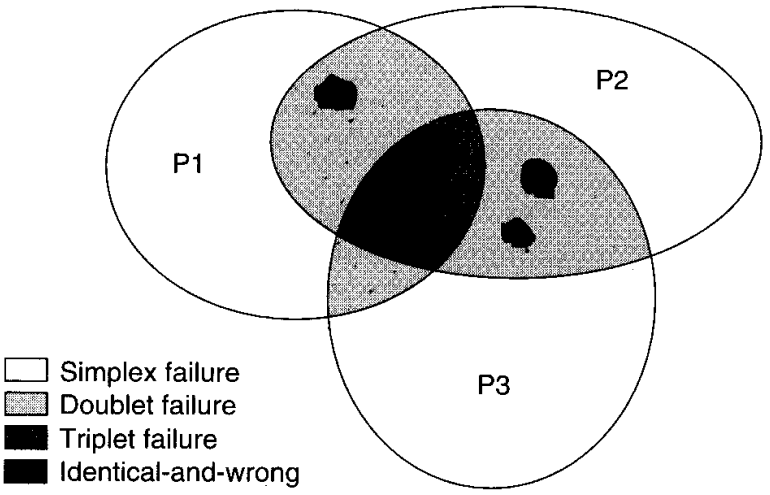


Figure 14.1 Illustration of failure space for three functionally equivalent programs.

components fail coincidentally. Of special interest are regions marked in black, which represent events where components produce IAW responses.

14.3.4 Adjudication by voting

A common adjudication algorithm is voting. There are many variants of voting algorithms (e.g., [Lorc89, McAl90, Vouk90b, Gers91, Lyu95a]). A voter compares results from two or more functionally equivalent software components and decides which, if any, of the answers provided by these components is correct.

14.3.4.1 Majority voting. In an m-out-of-N fault-tolerant software system, the number of versions is N (an N-tuple), and m is the *agreement number,* or the number of matching outputs which the adjudication algorithm (such as voting) requires for system success [Eckh85, Triv82, Siew92]. The value of N is rarely larger than 3. In general, in *majority voting,* $m = \lceil (N + 1)/2 \rceil$, where $\lceil\ \rceil$ denotes the ceiling function.

14.3.4.2 Two-out-of-*N* voting. It is shown in [Scot87] that, if the output space is large, and true statistical independence of variant failures can be assumed, there is no need to choose m larger than 2, regardless of the size of N. We use the term *2-out-of-N voting* for the case where agreement number is $m = 2$.

There is obviously a distinct difference between *agreement* and *correctness.* For example, a majority voter assumes that if a majority of the module outputs agree, then the majority output must be the correct output. This, however, can lead to fallacious results, particularly in the extreme case when the number of possible module outputs is very small. For example, suppose that the output of each module is a single variable that assumes the values of either 0 or 1. This means that all incorrect outputs automatically agree and it is very likely, if modules have faults, that there may be a majority of IAW outputs.

In general, there will be multiple output variables, each one assuming a number of values. The total number of allowed combinations of output variables and their values defines the number ρ of program output states, or the cardinality of the output space. If there are multiple correct outputs, then a simple voter is useless. Hence, when voting, we will assume that there is only one correct output for each input. From this it follows that, given output cardinality of ρ, we have one correct output state, and $\rho - 1$ error states.

In that context, mid-value selection (or simple *median voting*) is an interesting and simple adjudication alternative where the median of all output values is selected as the correct answer. The philosophy behind

the approach is that, in addition to being fast, the algorithm can handle multiple correct answers (and for small samples is less biased than averaging, or mean value voting), and it is likely to pick a value that is at least in the correct range. This technique has been applied successfully in aerospace applications.

14.3.4.3 Consensus voting. A generalization of majority voting is *consensus voting* described in [McAl90]. In consensus voting the voter uses the following algorithm to select the *correct* answer:

- If there is a majority agreement ($m \geq \lceil (N+1)/2 \rceil$, $N > 1$), then this answer is chosen as the correct answer.
- Otherwise, if there is a unique maximum agreement, but this number of agreeing versions is less than $\lceil (N+1)/2 \rceil$, then this answer is chosen as the correct one.
- Otherwise, if there is a tie in the maximum agreement number from several output groups, then

  If consensus voting is used in N-version programming, one group is chosen at random and the answer associated with this group is chosen as the correct one.

  Else if consensus voting is used in consensus recovery block, all groups are subjected to an acceptance test, which is then used to choose the correct output.

The consensus voting strategy is particularly effective in small output spaces because it automatically adjusts the voting to the changes in the effective output space cardinality. It can be shown that, for $m \geq 2$, majority voting provides an upper bound on the probability of failing the system using consensus voting, and 2-out-of-N provides a lower bound [McAl90]. When the output space cardinality is 2, the strategy is equivalent to majority voting, and to 2-out-of-N voting when the output space cardinality tends to infinity, provided the agreement number is not less than 2. We provide experimental comparison of consensus and majority voting strategies in Sec. 14.7.

14.3.5 Tolerance

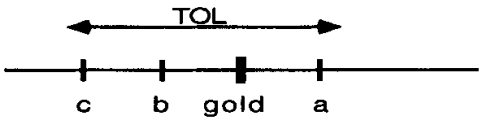
Closely related to voting is the issue of tolerance to which comparisons are made. Let TOL be the comparison tolerance, and consider an N-tuple of versions. The following two mutually exclusive events do *not* depend on whether the answers are correct.

- All N components agree on an answer. In this case we have an AGREEMENT event.

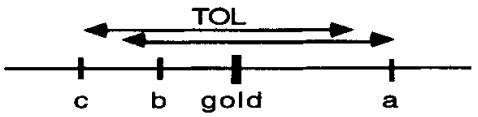
- There is at least one disagreement among the C_2^N comparisons of alternate outputs, where C_2^N denotes the number of combinations of N objects taken two at a time (2-tuples). We will call this case a CONFLICT event. All 2-tuples need to be evaluated because agreement may not be transitive, that is, $|a - b| \leq$ TOL and $|b - c| \leq$ TOL does not always imply that $|a - c| \leq$ TOL (see Fig. 14.2).

It is very important to realize that use of an inappropriate tolerance value, TOL, may either completely mask failure events (that is, too large a tolerance will always return AGREEMENT events) or cause an avalanche of CONFLICT events (the tolerance is too small and therefore many answer pairs fail the tolerance test). Assume that we have an oracle, so that we can tell the correctness of an answer. Note an oracle can take the form of an external consistency check (see Sec. 14.3.1.2). The following mutually exclusive events *depend on* the knowledge of the correctness of the output, or agreement with the correct answer:

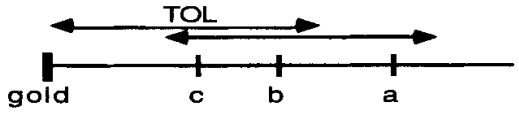
- All N components agree with the *correct* (call it *golden*) answer. Then a NO_FAILURE event occurs.
- One or more of the versions disagree with the *correct* answer. Then a FAILURE event occurs.



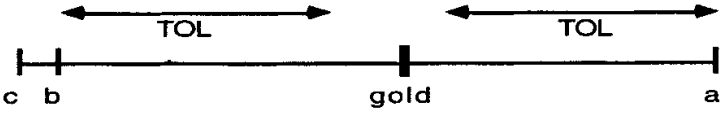
ALL_CORRECT is NO_FAILURE with AGREEMENT



FALSE_ALARM is NO_FAILURE with CONFLICT (|a-c|>TOL)



UN_DETECTED_FAILURE (|a-gold|>TOL) is FAILURE with AGREEMENT



DETECTED_FAILURE is FAILURE with CONFLICT (|a-b|>TOL, |a-c|>TOL)

Figure 14.2 Illustration of comparison events.

Combinations of the above elementary events produce the following mutually exclusive and collectively exhaustive multiversion comparison events:

- *ALL_CORRECT event.* A NO_FAILURE occurs with an AGREEMENT.
- *FALSE_ALARM event.* A NO_FAILURE occurs with a CONFLICT. Comparison signals an error when one is not present, which may or may not lead to a failure of the adjudication algorithm. Recall that agreement is not transitive, so FALSE_ALARM events are not inconsistent.
- *DETECTED_FAILURE event.* A FAILURE occurs together with a CONFLICT. Comparison correctly detects a failure (fault).
- *UN_DETECTED_FAILURE event.* A FAILURE event occurs simultaneously with an AGREEMENT. This is the most significant event. A potential failure exists but is not detected by comparison.

Consider again Fig. 14.1, which shows responses from a hypothetical three-version system. A simplex failure occurs when only one of the three versions fails (unshaded regions). A doublet failure (2-tuple) occurs when two components fail coincidentally (light shading). A triplet failure, or 3-tuple failure, occurs when all three components fail coincidentally (dark shading). If the probability that any of the *shaded* areas exceed or do not achieve the probability of overlap expected by random chance, then the assumption of independence is violated [Triv82]. That is,

$$P\{\mathrm{p}_i \text{ fails } \textit{and } \mathrm{p}_j \text{ fails}\} \neq P\{\mathrm{p}_i \text{ fails}\}\, P\{\mathrm{p}_j \text{ fails}\} \qquad (i \neq j) \tag{14.7}$$

or

$$P\{\mathrm{p}_1 \text{ fails } \textit{and } \mathrm{p}_2 \text{ fails } \textit{and } \mathrm{p}_3 \text{ fails}\} \neq P\{\mathrm{p}_1 \text{ fails}\}\, P\{\mathrm{p}_2 \text{ fails}\}\, P\{\mathrm{p}_3 \text{ fails}\} \tag{14.8}$$

where p_i is the ith software variant. The most undesirable state of the system, UN_DETECTED_FAILURE, occurs when the responses from three coincidentally failing versions are identical, in which case mutual comparison of these answers does not signal that a failure has occurred.

Figure 14.2 provides an illustration of the events that may result from comparison to tolerance TOL of floating-point outputs *a, b,* and *c,* from hypothetical programs p_1, p_2, and p_3, respectively. Additional examples are found in Prob. 14.5.

It is important to remember that excessively small tolerances may produce an excessive incidence of FALSE_ALARM events which may increase testing costs [Vouk88], while in operation this may result in degraded system operation or even a critical system failure.

14.4 Basic Techniques

Two common fault-tolerant software schemes are the N-version programming [Aviz77, Aviz85] and the recovery block [Rand75]. Both schemes are based on software component redundancy and the assumption that coincident failures of components are rare, and when they do occur responses are sufficiently dissimilar that the mechanism for deciding answer correctness is not ambiguous.

Fault-tolerant software mechanisms based on redundancy are particularly well suited for parallel processing environments where concurrent execution of redundant components may drastically improve sometimes prohibitive costs associated with their serial execution [Vouk90b, Bell90, Lapr90a]. Some hybrid techniques, such as consensus recovery block [Scot83, Scot87], checkpoints [Aviz77], community error recovery [Tso86, Tso87], N-version programming variants [Lapr90a, Tai93], and some partial fault-tolerance approaches are also available [Hech79, Stri85, Mili90].

14.4.1 Recovery blocks

One of the earliest fault-tolerant software schemes that used the multiversion software approach is the *recovery block* (RB) [Rand75, Deb86]. The adjudication module is an *acceptance test* (AT). The process begins when the output of the first module is tested for acceptability. If the acceptance test determines that the output of the first module is not acceptable, it restores, recovers, or "rolls back" the state of the system before the first or primary module was executed. It then allows the second module to execute and evaluates its output, etc. If all modules execute and none produce acceptable outputs, then the system fails. Figure 14.3 illustrates the technique.

One problem with this strategy in a uniprocessor environment is the sequential nature of the execution of versions [Lapr90a, Bell91], although in a distributed environment the modules and the acceptance tests can be executed in parallel. Distributed recovery block is discussed in detail in [Kim89, Lyu95a]. Another potential problem is finding a simple and highly reliable acceptance test which does not involve the development of an additional software version.

The form of acceptance test depends on the application. As suggested in Fig. 14.3, there may be a different acceptance test for each module,

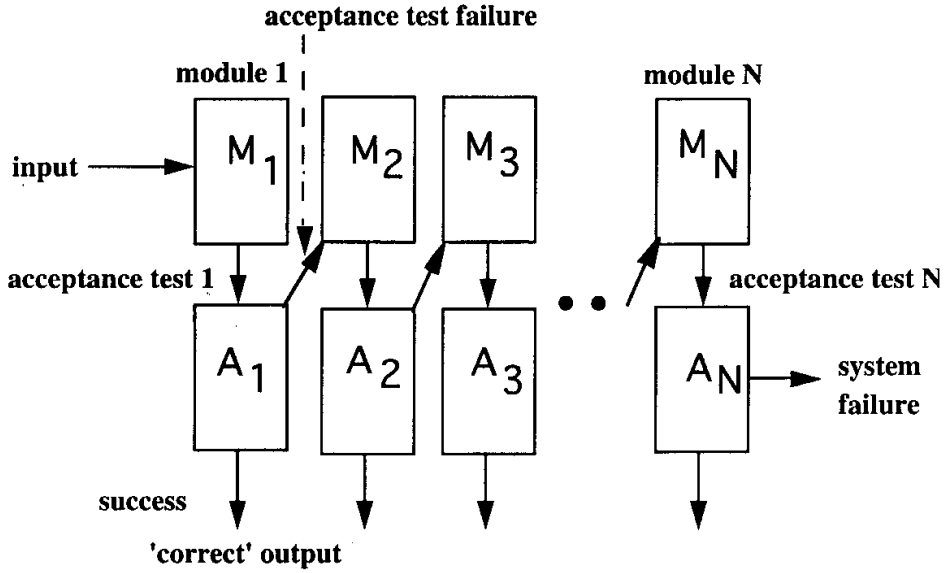


Figure 14.3 Recovery blocks.

but in practice only one is usually used. An extreme case of an acceptance test is another complete module, and the acceptance test would then consist of a comparison of a given module output with the one computed by the acceptance test. This would be equivalent to a staged two-version programming scheme (see the following section) where one of the outputs in each stage was always from the same version (the acceptance test).

14.4.2 N-version programming

N-version programming (NVP) [Aviz77, Chen78, Aviz85] proposes parallel execution of N independently developed functionally equivalent versions with adjudication of their outputs by a voter. N-version programming or multiversion programming (MVP) is a software generalization of the N-modular-redundancy (NMR) approach used in hardware fault tolerance [Nels87]. The N versions produce outputs to the adjudicator, which in this case is a *voter.* The voter accepts all N outputs as inputs and uses these to determine the correct, or best, output if one exists. There is usually no need to interrupt the service while the voting takes place. Figure 14.4 illustrates the technique. We note that the approach can also be used to help during testing to debug the versions. This method, called *back-to-back testing,* is discussed further in Sec. 14.8.2.

Over the years simple majority-voting-based N-version fault-tolerant software has been investigated by a number of researchers, both theoretically [Aviz77, Grna80, Eckh85, Scot87, Deb88, Litt89, Voge87a, Kano93a, Lyu95a] and experimentally [Scot84a, Scot84b, Bish86,

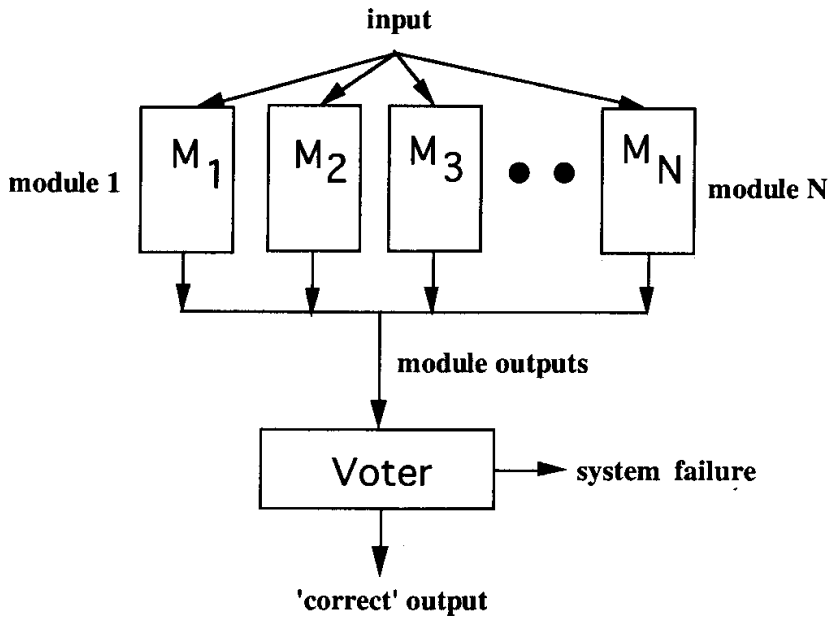


Figure 14.4 N-version programming.

Knig86, Shim88, Eckh91, Lyu93a, Duga93b, Vouk93a]. A reasonable alternative to majority voting could be to use consensus voting, described earlier. Another simple variation is *median voting.*

There are variants of the above architecture for distributed systems, some of which are further discussed in Chap. 15. One such variant is the *N self-checking programming* (NSCP), discussed below.

14.5 Advanced Techniques

There are ways to combine the preceding simple techniques to create hybrid techniques. Studies of more advanced models such as consensus recovery block [Bell90, Scot87, Deb88, Scot84a], consensus voting [McA190], or acceptance voting [Bell90, Atha89, Gant91, Gers91] are less frequent and mostly theoretical in nature.

14.5.1 Consensus recovery block

In [Scot83] and [Scot87] a hybrid system called *consensus recovery block* (CRB) was suggested. It combines NVP and RB in that order. If NVP fails the system reverts to RB using the same modules (the same module results can be used, or modules may be rerun if a transient failure is suspected). Only in the case that NVP and RB both fail does the system fail. A system block diagram is given in Fig. 14.5 where the block RB means that an acceptance test is applied to the outputs of the variants in the NVP block. CRB was originally proposed to treat

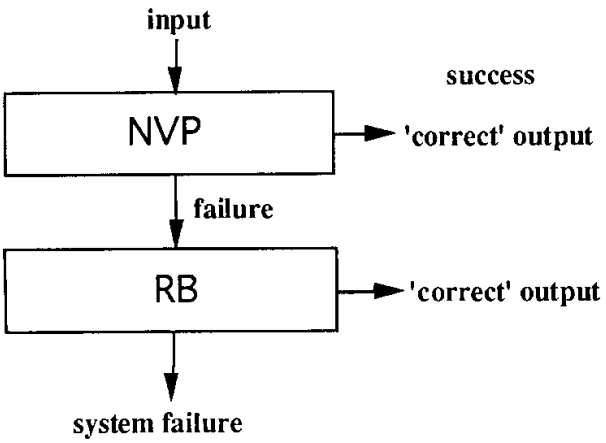


Figure 14.5 Consensus recovery block.

the case of multiple correct outputs, since appropriate acceptance testing can avoid that issue.

14.5.2 Acceptance voting

The converse of the above CRB hybrid scheme, which we call *acceptance voting* (AV), was proposed by [Atha89, Gant91, Bell91]. As in NVP, all modules can execute in parallel. The output of each module is then presented to an acceptance test. If the acceptance test accepts the output it is then passed to a voter. The system is shown in Fig. 14.6.

The voter sees only those outputs which have been passed by the acceptance test. This implies that the voter may not process the same number of outputs at each invocation and hence the voting algorithm must be dynamic. The system fails if no outputs are submitted to the voter. If only one output is submitted, the voter must assume it to be correct, and therefore passes it to the next stage. Only if two or more outputs agree can the voter be used to make a decision. We then apply dynamic majority voting (DMV) or dynamic consensus voting (DCV). The difference between DMV and MV is that even if a small number of results are passed to the voter, dynamic voting will try to find the majority among them. The concept is similar for DCV and CV.

14.5.3 N self-checking programming

A variant of the N-version programming with recovery, that is, N self-checking programming, is used in the Airbus A310 system [Lapr90a, Duga93b]. In NSCP, N modules are executed in pairs (for an even N) [Lapr90a]. The outputs from the modules can be compared, or can be assessed for correctness in some other manner. Let us assume that comparison is used. Then the outputs of each pair are tested and if they do not agree with each other, the response of the pair is dis-

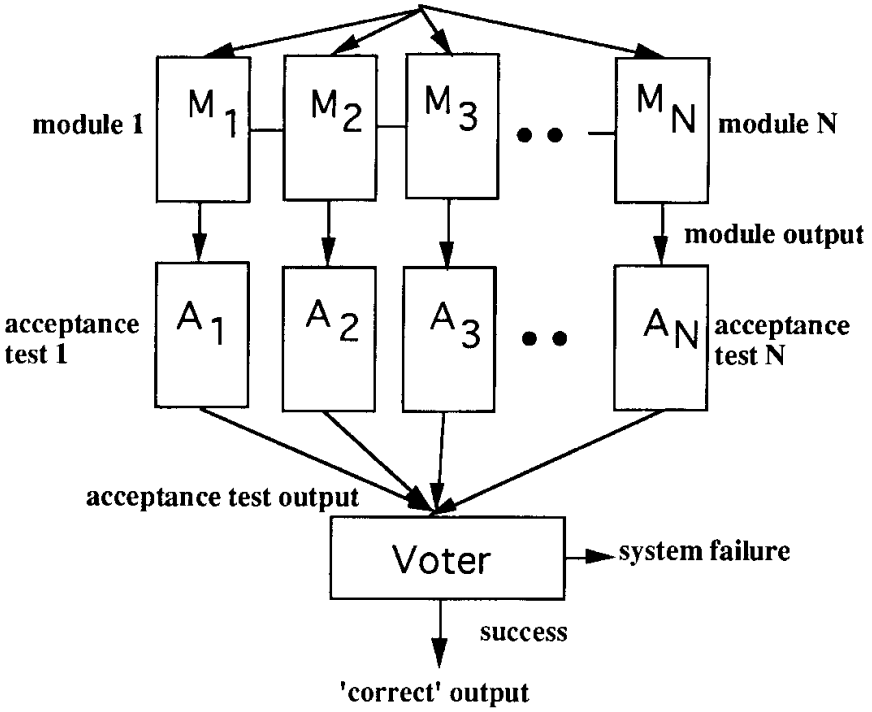


Figure 14.6 Acceptance voting.

carded. If the outputs of both pairs agree, then these outputs are compared again. Failure occurs if both pairs disagree, or the pairs agree but produce different outputs. The technique is shown in Fig. 14.7 for $N = 4$.

If a comparison of the outputs of the first pair of modules, M_1 and M_2, is successful, then the output is passed to the next phase of the computation and the system is successful. If these two outputs disagree, then a comparison of the outputs of the second pair of modules, M_3 and M_4, is made. If the outputs of the second pair agree, then the output is passed to the next phase. Otherwise the system fails. There is no attempt to compare the four outputs simultaneously. We leave an analysis of the reliability of the system as exercise.

14.6 Reliability Modeling

Although existing fault-tolerant software (FTS) techniques can achieve a significant improvement over non-fault-tolerant software, they may not be sufficient for ensuring adequate reliability of critical systems. For example, experiments show that incidence of correlated failures of FTS system components may not be negligible in the context of current software development and testing techniques, and this may result in a disaster [Scot84a, Vouk85, Eckh85, Knig86, Kell86, Kell88, Eckh91]. Obviously, it is important to detect and eliminate faults causing dependent failures as early as possible in the FTS life cycle and to develop

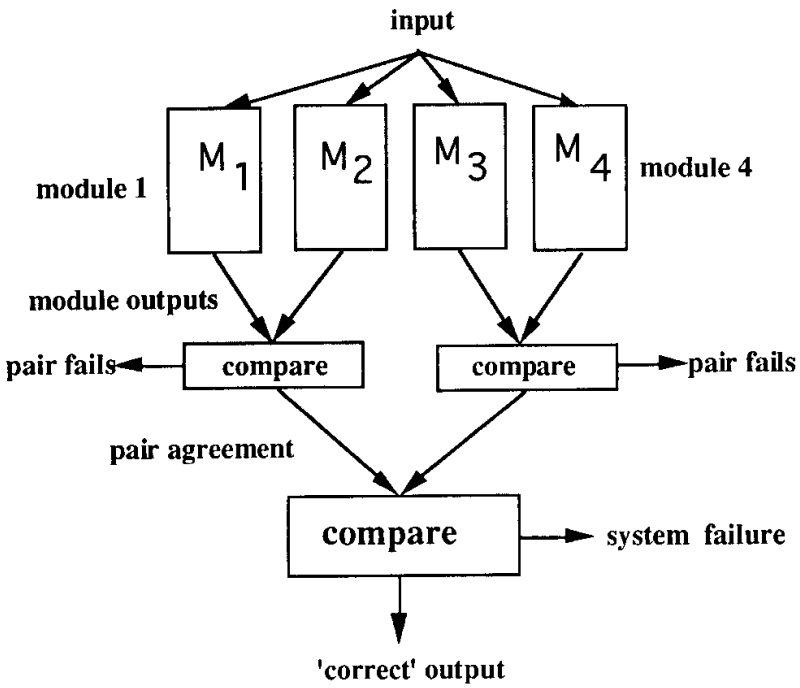


Figure 14.7 NSCP for $N = 4$.

FTS mechanisms that can adequately cope with any remaining faults. Modeling of different FTS mechanisms gives insight into their behavior and allows quantification of their relative merits.

14.6.1 Diversity and dependence of failures

Intuition suggests that the modules which are used to compose a multiversion fault-tolerant system should be as diverse as possible. Software reliability cannot be improved by using similar versions in a multiversion system. The issue is how to quantify diversity in versions and which techniques can be used to encourage the diversity [Lyu94a].

We first distinguish between specific inputs and programs versus randomly chosen inputs and programs. Hence, suppose a countable collection of program versions PV = $\{p_1, p_2, \ldots\}$ have been independently developed, and assume, for generality, that the probability of selecting one of the programs from PV is governed by the random variable Π with density function S. That is, $S(p)$ is the probability that p is chosen from PV or that $\Pi = p$. Also assume that the possible set of inputs is $\mathbf{X} = \{x_1, x_2, x_3, x_4, \ldots\}$ and that the probability that an arbitrarily chosen input $X = x$ is the value of the density function Q, $Q(x)$. Let v be the bivariate *score* function v defined by

$$v(p_i, x_j) = 1 \quad \text{if program } p_i \text{ fails on input } x_j \qquad (14.9)$$

$$= 0 \quad \text{otherwise}$$

The value $v(\mathrm{p}_i, x_j)$ is the probability that the specific program p_i fails on the specific input x_j. Let us first calculate the probability that an arbitrary program Π fails on a given input x:

$$\begin{aligned} P\{\Pi \text{ fails on } x\} &= \sum_{\mathrm{PV}} v(\mathrm{p},x)P\{\Pi = \mathrm{p}\} \\ &= \sum_{\mathrm{PV}} v(\mathrm{p},x)S(\mathrm{p}) \\ &= \theta(x) \end{aligned} \tag{14.10}$$

The function θ is the *intensity function* defined in [Eckh85]. Similarly, we can define the probability that a specific program p fails on an arbitrary input X [Litt89]:

$$\begin{aligned} P\{\mathrm{p} \text{ fails on } X\} &= \sum_{X} v(\mathrm{p},x)P\{X = x\} \\ &= \sum_{X} v(\mathrm{p},x)Q(x) \\ &= \phi(\mathrm{p}) \end{aligned} \tag{14.11}$$

Correspondingly, we can also define the probability that an arbitrary program Π fails on an arbitrary input X. Because of the duality of the previous two probabilities we can define the random variable $\theta(X) = \Theta$, or the random variable $\phi(\Pi) = \Phi$ and take their expected values ($E[\,]$) over the appropriate domain:

$$\begin{aligned} P\{\Pi \text{ fails on } X\} &= E[\Theta] = E[\Phi] \\ &= \sum_{\mathrm{PV}} \sum_{X} v(\mathrm{p},x)S(\mathrm{p})Q(x) \end{aligned} \tag{14.12}$$

We now wish to compute the probability that *two* randomly chosen programs fail on a single randomly chosen input X. This probability becomes

$$\begin{aligned} &P\{\Pi_1 \text{ fails on } X \text{ and } \Pi_2 \text{ fails on } X\} \\ &= \sum_{X} \sum_{\mathrm{PV}} \sum_{\mathrm{PV}} v(\mathrm{p}_1,x)v(\mathrm{p}_2,x)S(\mathrm{p}_1)S(\mathrm{p}_2)Q(x) \\ &= \sum_{X} (\theta(x))^2 Q(x) = E[\Theta^2] \\ &= \sigma_{\Theta}^2 + E^2[\Theta] \end{aligned} \tag{14.13}$$

where σ_Θ^2 is the variance of the random variable Θ. The variance is zero if and only if $\theta(x)$ is equal to its expected value for all x, that is, θ must be a constant for two random programs to fail independently on a random input!

However, we are interested in specific programs failing on random inputs. The probability that two specific programs, p_1 and p_2, fail on an arbitrary input X (i.e., the probability that p_1 and p_2 have coincident failures) is

$$P\{p_1 \text{ fails on } X \text{ and } p_2 \text{ fails on } X\} = \sum_X v(p_1,x)v(p_2,x)Q(x) \quad (14.14)$$

If the failures were independent then this product should be equal to $P\{p_1 \text{ fails on } X\}P\{p_2 \text{ fails on } X\}$. We leave it as an exercise to show that the latter product can underestimate *or* overestimate joint failure depending on the amount of overlap of the input failure sets of the two programs p_1 and p_2. As you would expect, in practice, we wish for the failure input sets to be disjoint to minimize the probability of coincident failures. If this is the case, the independence assumption overestimates the probability of joint failure.

Scott and his colleagues [Scot84b] were first to show, using an experiment, that programs may not fail independently, and they developed models to treat this case. Their results were later corroborated by other experimenters. Unfortunately, the models developed to treat the general case become quite complicated and intractable as the number of versions increases, since joint failure probabilities must be estimated. A set of experimental data related to failure correlation is discussed later in this chapter. For a more general treatment of the correlation issue, you should consult the works by [Scot84a, Eckh85, Litt89].

Originally, the definition of reliability involved the behavior of hardware over time. We are also interested in software behavior over time as faults are identified and repaired. In critical systems, however, we are interested also in the behavior of redundant modules for each input and we wish to know the probability that a software-fault-tolerant system will produce the correct answer. This motivates the approach called *data-domain reliability modeling* [Ande81]. We can then use this and other information, as well as techniques such as Markov modeling and Petri nets, to determine such standard parameters as mean time to failure, etc. The latter is part of the time-domain analysis. Time-domain analysis is concerned with the behavior of FTS systems over a mission (or time period) within which reliability of individual components may or may not change. It is discussed very briefly later in this section and, for example, in [Kano93a].

The *reliability* of a module is the probability it will produce the correct output (assuming the input is correct) where inputs are taken from the operational input domain with density function Q. Since Q is rarely known, it is usually assumed to be uniformly distributed or random. Normally, the *reliability* of a module is estimated from testing and is a function of the number of test cases that have been presented.

Let f_i denote the probability that module p_i fails (on an arbitrary input):

$$f_i = P\{\text{module } p_i \text{ fails}\} = \sum v(p_i, x)Q(x) \tag{14.15}$$

An estimate of the failure probability, $\hat{f}_i$, is the number of failures observed in presenting random inputs to a module based on the operational input distribution. That is, if k is the number of failures in n random inputs, then

$$\hat{f}_i = \frac{k}{n} \tag{14.16}$$

We leave it to the reader to estimate the variance of this estimate. We will assume that we have a nonzero value for $\hat{f}_i$ in the discussion below. In the independent case, the reliability of module i, that is, the probability that it will not fail, is estimated as

$$\hat{r}_i = 1 - P\{\text{module } i \text{ fails}\} = 1 - \hat{f}_i \tag{14.17}$$

14.6.2 Data-domain modeling

The following data-domain analysis examples are intended to illustrate two things: (1) how to construct models by first defining *events* of interest and the associated probabilities, and then combining them into a reliability estimate, and (2) provide insight into the relative merits of different fault-tolerance strategies.

For tractability, the analyses are made using the *assumption* that intervariant failure events are independent, i.e., the failures of specific programs on random inputs are independent of each other. Although this abstraction simplifies the modeling and provides insight into the relative behavior of different FTS strategies, it does not always provide a realistic result for real-life situations where common-cause faults and correlated coincident failures are present. For analyses that incorporate different interversion failure correlation assumptions, the reader is directed to works of [Eckh85, Litt89, Nico90, Tai93, Kano93a, Tome93, Duga94c, Duga95, Lyu95a].

We restrict our examples to $N = 3$, and we leave it as exercises to analyze the systems for other values of N. We also leave it to the reader to

construct solutions for more complex mechanisms (e.g., AV and CRB, see Probs. 14.18 and 14.19). In this context, let r_1, r_2, and r_3 be the reliabilities of each version of a three-version fault-tolerant system. Let B be the reliability of the acceptance test in RB; let R_V be the reliability of the voter in NVP; and let S denote the system reliability.

Recovery block. In RB, to simplify analysis, we assume that the (conditional) probability, β, of rejecting a correct answer is equal to the probability of accepting an incorrect one, that is $\beta = (1 - B)$. Then the system success depends on at least one module producing correct output and the acceptance test recognizing that it is correct. We can partition the event space by the number of the module which produces the correct output that is also accepted by the AT. We will assume that the state of the system is always recovered without error. Let the three-version RB be denoted by RB3.

1. The first event is that the primary module is correct and the AT accepts the output. The probability of this event is r_1B.
2. The second event is that the output of the first module is rejected and the output of the second module is correct and accepted. This event has two possible outcomes: the first module can be incorrect and the AT appropriately rejects it, or the first module can be correct but the output is rejected by the AT. Hence, the probability of the second event is $(1 - r_1)\, B\, r_2\, B + r_1\, (1 - B)\, r_2\, B$.
3. Similarly, for the third event, if the third module produces the correct output, then the output of modules 1 and 2 were rejected by the AT. This can happen in one of four ways:
   a. Module 1 output is incorrect and the AT correctly rejects it, module 2 output is incorrect and the AT correctly rejects it, and module 3 is correct and the AT accepts it.
   b. Module 1 output is correct but the AT rejects it, module 2 output is incorrect and the AT correctly rejects it, and module 3 output is correct and the AT correctly accepts it.

   Cases *c* and *d* are similar.

Rearranging and simplifying the sum of these probabilities yields a system reliability of

$$S_{RB3}(r_1, r_2, r_3, B) = B(r_1 + r_1r_2 + r_1r_2r_3 + r_2B - 2r_1r_2B + r_1r_3B + r_2r_3B - 4r_1r_2r_3B + r_3B^2 - 2r_1r_3B^2 - 2r_2r_3B^2 + 4r_1r_2r_3B^2 \quad (14.18)$$

If we assume that all versions have the same reliability, r, we have

$$S_{RB3}(r, r, r, B) = B\, r\, (1 + B + B^2 + r + r^2 - 4r^2B - 4rB^2 + 4r^2B^2) \quad (14.19)$$

If, in addition, we assume $B = 1$, that is, ideal operation of the acceptance testing algorithm, Eq. (14.19) reduces to

$$S_{RB3}(r, r, r, 1) = 3\,r\,(1 - r) + r^3 \tag{14.20}$$

The surface plot of RB3 system reliability as a function of B and r is shown in Fig. 14.8.

N-version programming. The reliability of a three-version NVP system is the probability that at least two modules produce correct output and that the voter is correct. The probability that at least two modules are correct is the probability that modules 1 and 2 are correct and module 3 fails, or modules 1 and 3 are correct and module 2 fails, or modules 2 and 3 are correct and module 1 fails, or that all three are correct. Under our assumptions, this probability is

$$r_1r_2(1 - r_3) + r_1r_3(1 - r_2) + r_2r_3(1 - r_1) + r_1r_2r_3 \tag{14.21}$$

Hence, if we assume voter failure is also independent of module failure, the system reliability $S_{NVP3}(r_1, r_2, r_3, R_V)$, of a three-version NVP fault-tolerant system becomes

$$S_{NVP3}(r_1, r_2, r_3, R_V) = R_V\,(r_1r_2 + r_1r_3 + r_2r_3 - 2r_1r_2r_3) \tag{14.22}$$

In practice we would expect that all units that comprise an FTS system will have been tested to the point where there are *no known* residual faults. This may or may not mean that they have very similar reliability, but it may be a reasonable assumption. Hence, an interesting special case is the one where all functionally equivalent versions are assumed to have equal, and very high, lower bounds on their relia-

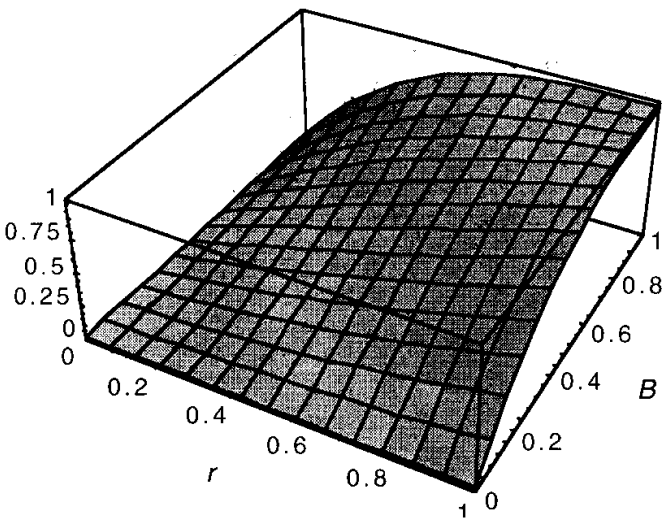


Figure 14.8 S_{RB3} as functions of r and B.

bility. Individual version reliability estimates must be obtained independently using, for example, appropriate reliability growth models.

Let all versions have the same estimated reliability r. Then the probability that an NVP system will operate successfully (assuming a perfect voter) under majority voting strategy is given by the following expression (e.g., [Grna80, Triv82, Nels87]):

$$S_{\text{NVP3}}(r_1 = r_2 = \ldots = r_n = r, R_V) = R_V \sum_{i=m}^{n} C_i^n r^i (1-r)^{n-i} \qquad (14.23)$$

where summation starts with the lower bound on required agreement number.

Equation (14.23) can be used to show that majority voting increases reliability over a single version only if the reliability of the versions is larger than 0.5 and the voter is perfect. If the output space has cardinality ρ, then N-version programming will result in a system that is more reliable than a single component only if $r > 1/\rho$ [McAl90].* We call this value the *boundary version reliability.* It is the generalization of the classical N-modular redundancy rule for a binary output space where $r > 0.5$, and it applies in the case of consensus voting. Note that, when a version fails and we let the probability of occurrence of any incorrect output be q, then, in the simplest situation, $q = (1 - r)/(\rho - 1)$.

In Fig. 14.9 we show the classical majority voting approach with a binary output space (boundary version reliability of $1/\rho = 0.5$). We see that the version reliability must be larger than the boundary version reliability in order to improve the performance of the system when more versions are added.

Figure 14.10 shows the effect of version reliability and the number of versions under consensus voting strategy. The minimal agreement number is $m = \lfloor (n + \rho - 1)/\rho \rfloor = \lfloor (n + 2)/3 \rfloor$, where $\lfloor\ \rfloor$ denotes the floor function. The average boundary reliability of the versions is $1/\rho = 1/3$. All versions are assumed to have the same reliability, and all failure states ($j = 2,3$) the same probability $(1 - q)/(1 - \rho) = (1 - q)/2$ of being excited.

If all versions have the same reliability we have

$$S_{\text{NVP3}}(r, r, r, R_V) = R_V(3r^2 - 2r^3) \qquad (14.24)$$

We note that S_{NVP3} is bounded by R_V. Hence, if $R_V \leq r$, then one should opt to invest software development time on a single version rather than develop a three-version NVP system.

---

* Additional assumptions: all components fail independently; they have the same reliability r; correct outputs are unique; and the voter is perfect.

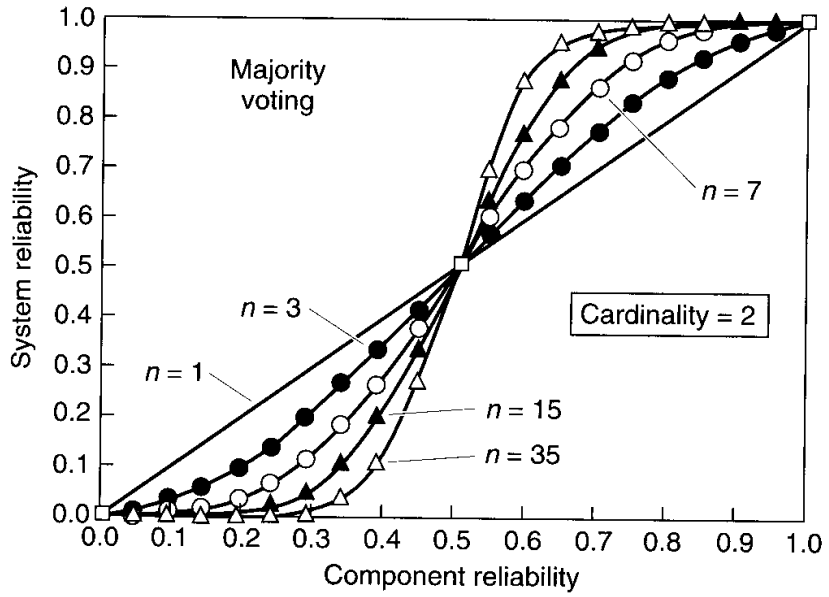


Figure 14.9 System reliability versus component reliability for majority voting strategy. Number of components used for voting is n, the agreement number is m, $\rho = 2$, and boundary version reliability is $1/\rho = 0.5$.

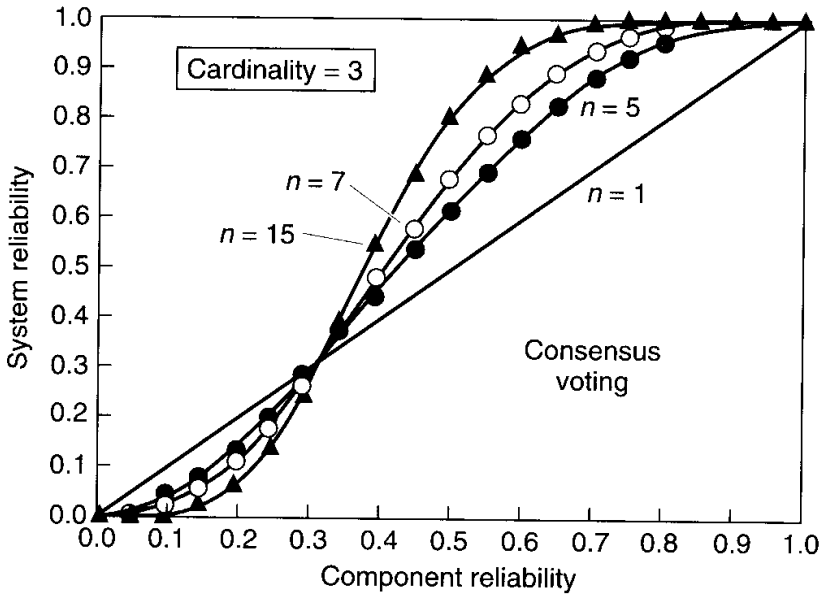


Figure 14.10 System reliability versus component reliability for the consensus voting strategy. The number of voting components is n, all are equally reliable, the agreement number is m, $\rho = 3$, and the boundary version reliability is $1/\rho = 0.3333$.

It is interesting to compare an RB3 with perfect acceptance test to an NVP3 system with perfect voter. This is done in Fig. 14.11. We see that under the assumption of failure independence, the RB system is a better solution than a majority-voting three-version system. However, note that it may be far easier to ensure very high reliability of the voting software than to devise and implement an acceptance test with no faults.

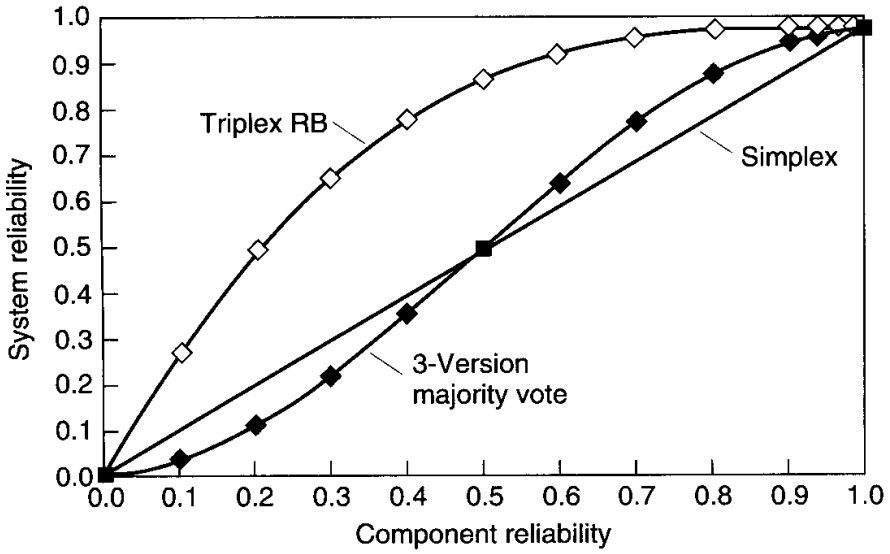


Figure 14.11 Comparison of N-version and recovery block schemes for $n = 3$. Both voter and acceptance test are assumed to be perfect.

This is confirmed by the work of Kanoun et al. [Kano93a]. They have shown that the impact of *independent* faults is much higher in the case of NVP than in the case of RB because the versions are run in parallel. They also concluded that the impact of common-cause faults will likely be higher in the case of RB than in the case of NVP because very reliable acceptance tests may be difficult to construct. The results shown in Sec. 14.7 confirm this. We note the more substantial impact of faults on NVP when reliability of individual components is low, and the better performance of NVP when reliability of components is higher but RB acceptance test is failure prone. In the case of RB, common-cause faults include *similar* or *related* faults among RB alternates and the acceptance test, as well as *independent faults* in the acceptance test. In the case of NVP, common-cause faults are similar or *related* faults among components and the voter, as well as *independent* faults in the voter.

14.6.3 Time-domain modeling

Time-domain analysis is concerned with the behavior of system reliability over time. For example, during software debugging and testing we would expect the reliability of the components (e.g., [Musa87]) and the system (e.g., [Kano93a]) to grow. On the other hand, during operation without repair, the reliability of a component remains constant. In the time-domain, reliability can be defined as the probability that a system will complete its mission, or operate through a certain period of time, without failing.

The simplest time-dependent failure model assumes that failures arrive randomly with interarrival times exponentially distributed with

expected value λ. Hence, the probability that a module will produce the correct output decreases over time since it receives a larger number of inputs, and we have

$$r(t) = e^{-\lambda t} \tag{14.25}$$

During operation it is often assumed that the *failure* or *hazard rate* λ is constant; however, during reliability growth or reliability decay periods the failure rate is itself a function of time, and then the expression that describes r becomes more complex (e.g., [Musa87, Kano93a]).

Time-dependent behavior of components can have significant impact on the operation of a fault-tolerant system. Therefore, it is important that practical fault-tolerant systems are analyzed not only with respect to their data-domain characteristics, but also with respect to their time-domain characteristics (e.g., [Grna80, Triv82, Deb86, Deb88, Arla90, Lapr90a, Siew92, Tai93, Kano93a]).

To illustrate, consider the following. Assuming that Eq. (14.25) holds, that λ is fixed, and that $R_V = 1$, Eq. (14.24) can be modified to yield

$$S_{\text{NVP3}}(r, r, r, 1, t) = 3e^{-2\lambda t} - 2e^{-3\lambda t} \tag{14.26}$$

This system has an interesting property. If we plot $S_{\text{NVP3}}(r, r, r, 1, t)$ and $r(t)$ against time, we find that the two curves cross when $t = t_0 = \ln 2/\lambda \approx 0.7/\lambda$ [Triv82, Siew92]. For $t \leq t_0$ the system is more reliable than a single version, but during longer missions, $t > t_0$, NVP3 fault tolerance may actually degrade reliability.

Of course, the above is just an illustration of a very special case. Usually, the problem is far more complex, and a complete analysis, including failure correlation effects and hardware issues, is essential (see Chap. 15 and [Grna80, Tome93, Kim89, Lyu95a, Duga95]). For example, Tomek et al. have modeled RB with failure correlation and have analyzed the time-dependent behavior of RB reliability in considerable detail [Tome93]. Another example is the work of Kanoun et al. [Kano93a], who have modeled reliability growth of individual components using the hyperexponential model [Lapr90b], and have analyzed the impact this has on reliability of NVP and RB models. As mentioned earlier, they have found that NVP is far more sensitive to the removal of independent faults than RB because of the parallel nature of the NVP execution and decision making (voting).

On the other hand, if similar or related faults are present they are likely to have a larger impact on RB performance because acceptance tests tend to be more complex and more correlated to the actual application-specific nature of the components than simple voting comparisons. Hence, removal of similar or related faults and faults in decision

nodes will probably produce more substantial reliability gains in the case of RB than in the case of NVP.

14.7 Reliability in the Presence of Interversion Failure Correlation

Experiments have shown that interversion failure dependence among independently developed functionally equivalent versions may not be negligible in the context of current software development and testing strategies [Scot84a, Scot84b, Vouk85, Knig86, Eckh91, Gers91]. There are theoretical models of FTS reliability which incorporate interversion failure dependence in different ways (e.g., [Eckh85, Litt89, Nico90, Tai93, Duga94c]). Coincident failures can be treated as resulting from statistically correlated faults (e.g., [Eckh85, Nico90]) or as deriving from *related* or *similar* faults that cause IAW results, and from unrelated or independent software faults that cause *dissimilar* but wrong results (e.g., [Lapr90a, Duga94c]). Furthermore, the related and unrelated software faults can be assumed to behave in a statistically independent [Duga94c], mutually exclusive [Lapr90a], or some other manner. However, many models assume interversion failure independence as well as failure independence of acceptance tests and voters with respect to versions and each other.

In this section we illustrate the effects that can be observed in an FTS systems under severe failure correlation conditions. We compare several experimental implementations of consensus recovery block and consensus voting with more traditional schemes, such as N-version programming with majority voting. The data derived from the experimental study is described in [Vouk93a].

14.7.1 An experiment

Experimental results based on a pool of functionally equivalent programs developed in a large-scale multiversion software experiment are described in several papers [Kell88, Vouk90a, Eckh91, Vouk93a]. Twenty versions of an avionics application were developed by 20 two-member development teams working independently at four universities. The versions were written in Pascal and ranged in size between 2000 and 5000 lines of code.

The results discussed here are for the program versions in the state they were in immediately after the unit development phase, but *before* they underwent an independent validation phase (in real situations, versions would be rigorously validated before operation). This was done (1) to keep the failure probabilities of individual versions relatively high and easier to observe and (2) to retain a considerable num-

ber of faults that exhibit mutual failure correlation in order to highlight correlation-based effects. The nature of the faults found in the versions is discussed in detail in two papers [Vouk90a, Eckh91].

Two subsets of N programs (called *N-tuples*) were generated: those with similar average* N-tuple reliability and those that have reliability within a particular range. The average N-tuple reliability is used to focus on the behavior of a particular N-tuple instead of the population (pool) from which it was drawn.

In the experiment a number of input profiles, different combinations of versions, and different output variables were considered. Failure probability estimates, based on the three most critical output variables (out of 63 monitored), are shown in Table 14.1. The variables are of type *real,* and each has a very large output space.

Two test suites, each containing 500 uniform random-input test cases, were used in all estimates. The sample size is sufficient for the version and N-tuple reliability ranges reported here. One suite, called Estimate I, was used to (1) estimate individual version failure probabilities, (2) N-tuple reliability, (3) select acceptance test versions, (4) select sample N-tuple combinations, and (5) compute expected independent model response. The other test suite, Estimate II, was used to investigate the actual behavior of N-tuple systems, based on different voting and fault-tolerance strategies.

For recovery block and consensus recovery block, one version was used as an acceptance test. This provided correlation not only among versions, but also between the acceptance test and the versions. Acceptance test versions were selected first, then N-tuples were drawn from the subpool of remaining versions. The voter (comparator) was assumed to be perfect.

The fault-tolerant algorithms of interest were invoked for each test case. The outcome was compared with the correct answer obtained

---

* Average N-tuple reliability estimate is defined as

$$\bar{p} = \sum_{i=1}^{N} \frac{\hat{p}_i}{N}$$

and the corresponding estimate of the standard deviation of the sample as

$$\hat{\sigma} = \sqrt{\sum_{i=1}^{N} \frac{(\bar{p} - \hat{p}_i)^2}{N-1}}$$

where

$$\hat{p}_i = \sum_{j=1}^{k} \frac{s_i(j)}{k}$$

is estimated reliability of version i over the test suite composed of k test cases, $s_i(j)$ is a score function equal to 1 when version i succeeds and 0 when it fails on test case j, and $1 - \hat{p}_i$ is the estimated version failure probability.

TABLE 14.1 Estimated Version Failure Probabilities ($\hat{f}_i$)

	Failure rate*	
Version	Estimate I	Estimate II
1	0.58	0.59
2	0.07	0.07
3	0.13	0.11
4	0.07	0.06
5	0.11	0.10
6	0.63	0.64
7	0.07	0.06
8	0.35	0.36
9	0.40	0.39
10	0.004	0.000
11	0.09	0.10
12	0.58	0.59
13	0.12	0.12
14	0.37	0.38
15	0.58	0.59
16	0.58	0.59
17	0.10	0.09
18	0.004	0.006
19	0.58	0.59
20	0.34	0.33

* Based on three most important output variables. Each column was obtained on the basis of a separate set of 500 random test cases.

from a "golden" program [Aviz77, Vouk90a], and the frequency of successes and failures for each strategy was recorded.

14.7.2 Failure correlation

The failure correlation properties of the versions can be deduced from their joint coincident failure profiles, and from the corresponding *identical-and-wrong* response profiles. For example, Tables 14.2 and 14.3 show the profiles for a 17-version subset (three versions selected to act as acceptance tests are not in the set). In Table 14.2 we show the number of versions that fail coincidentally, and the corresponding number of occurrences of the event over the 500 samples. Also shown is the expected number of occurrences for the model based on independent failures, or the binomial model [Triv82].

For instance, inspection of Table 14.2 shows that the number of occurrences of the event where nine versions fail coincidentally is expected to be about 8. In reality, we observed about 100 such events. Table 14.3 summarizes the corresponding occurrences of *identical-and-wrong* coincident responses. For example, in 500 tries there were 15 events where eight versions coincidentally returned an answer which

TABLE 14.2 Frequency Data for a 17-Version Set

No. of versions that failed together	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Frequency	168	13	28	0	0	0	75	31	14	105	15	17	0	18	15	1	0	0
Model	0	4	18	52	96	121	106	65	28	8	2	0	0	0	0	0	0	0

TABLE 14.3 Frequency of Empirical IAW Events Over 500 Test Cases for a 17-Version Set

No. of versions that coincidentally returned an IAW answer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Frequency	2049	164	1	16	1	1	2	15	0	0	0	0	0	0	0	0	0

was wrong yet identical within the tolerance used to compare the three most critical (real) variables. These results are strong indicators of a high degree of interversion failure dependence in this version set.

14.7.3 Consensus voting

Figures 14.12 and 14.13 illustrate the observed relationship between N-version programming with consensus voting and with majority voting. The figures show success frequency for three-version and seven-version systems over a range of average N-tuple reliability assuming perfect voting. The ragged look of the experimental traces is partly due to the small sample (500 test cases), and partly due to the presence of very highly correlated failures. The experimental behavior is in good agreement with the trends indicated by the theoretical consensus voting model based on failure independence [McAl90, Vouk93a].

For instance, we see that for $N = 3$ and low average N-tuple reliability, N-version programming has difficulty competing with the best version. Note that the *best version* was not preselected based on Estimate I data. Instead, it is the N-tuple version which exhibits the smallest number of failures during the actual evaluation run (Estimate II). The reason N-version programming has difficulty competing with the best version is that the selected N-tuples of low average reliability are composed of versions which are not balanced, that is, their reliability is very different, and therefore variance of the average N-tuple reliability is large. As average N-tuple reliability increases, N-version programming performance approaches or exceeds that of the best version. In part, this is because N-tuples become more balanced, since the number of higher-reliability versions in the subpool from which versions are

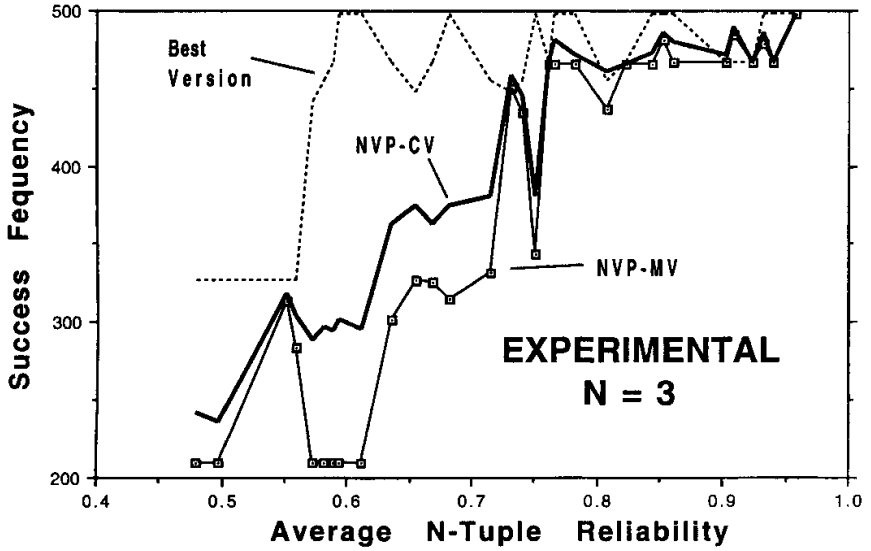


Figure 14.12 System reliability by voting ($N = 3$).

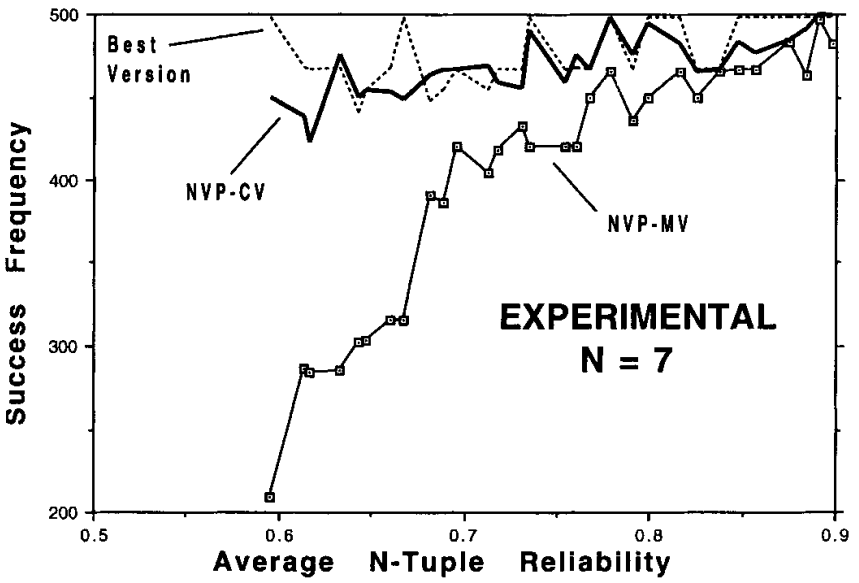


Figure 14.13 System reliability by voting ($N = 7$).

selected is limited. We also see that $N > 3$ improves performance of consensus voting more than it does that of majority voting. This is to a large extent because, for $N > 3$, *plurality* decisions become possible, that is, in situations where there is a unique maximum of identical outputs, the output corresponding to this maximum is selected as the correct answer even though it is not in the majority.

The advantage of consensus voting is that it is more stable than majority voting. It always offers reliability at least equivalent to majority voting, and it performs better than majority voting when average N-tuple reliability is low, or the average decision space in which voters work is not binary. A practical disadvantage of consensus voting may

be the added complexity of the voting algorithm, since the strategy requires multiple comparisons and random number generation.

14.7.4 Consensus recovery block

In the case of version failure independence and zero probability for identical-and-wrong responses, consensus recovery block is always superior to N-version programming (given the same version reliability and the same voting strategy), or to recovery block (given the same version and acceptance test reliability) [Scot87, Bell90]. However, given the same voting strategy, and very high interversion failure correlation, we would expect consensus recovery block to do better than N-version programming only in situations where coincidentally failing versions return different results. We would not expect the consensus recovery block to be superior to N-version programming in situations where the probability of identical-and-wrong answers is very high, since then many decisions are made in a very small voting space, and the consensus recovery block acceptance test is invoked very infrequently.

Figures 14.14 and 14.15 show the number of times the result provided by a fault-tolerance strategy was correct, plotted against the average N-tuple reliability. The same acceptance test version was used by consensus recovery block and recovery block. From Fig. 14.14 we see that for $N = 3$, consensus recovery block with majority voting provided reliability *always* equal to or better than the reliability by N-version programming with majority voting (using the same versions). The behavior of a five-version system using consensus voting, instead of majority voting, is shown in Fig. 14.15. From the figure we see that, at lower N-tuple reliability, N-version programming with consensus voting becomes almost as good as consensus recovery block. Consensus recovery block with consensus voting is quite successful in competing with the best version. However, it must be noted that, given a sufficiently reliable acceptance test, or binary output space, or very high interversion failure correlation, all schemes that vote may have difficulty competing with recovery block.

Consensus recovery block with consensus voting is a more advanced strategy than N-version programming with consensus voting, and most of the time it is more reliable than N-version programming with consensus voting. However, there are situations where the reverse is true. Consensus recovery block with consensus voting employs the acceptance test to resolve situations where there is no plurality. N-version programming with consensus voting uses random tie-breaking. N-version programming with consensus voting may be marginally more reliable than consensus recovery block with consensus voting

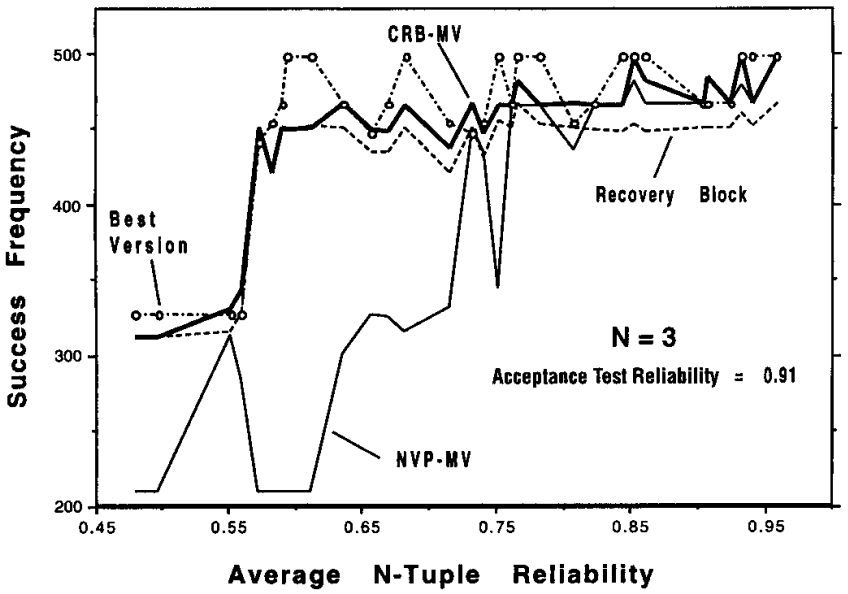


Figure 14.14 Consensus recovery block system reliability with majority voting.

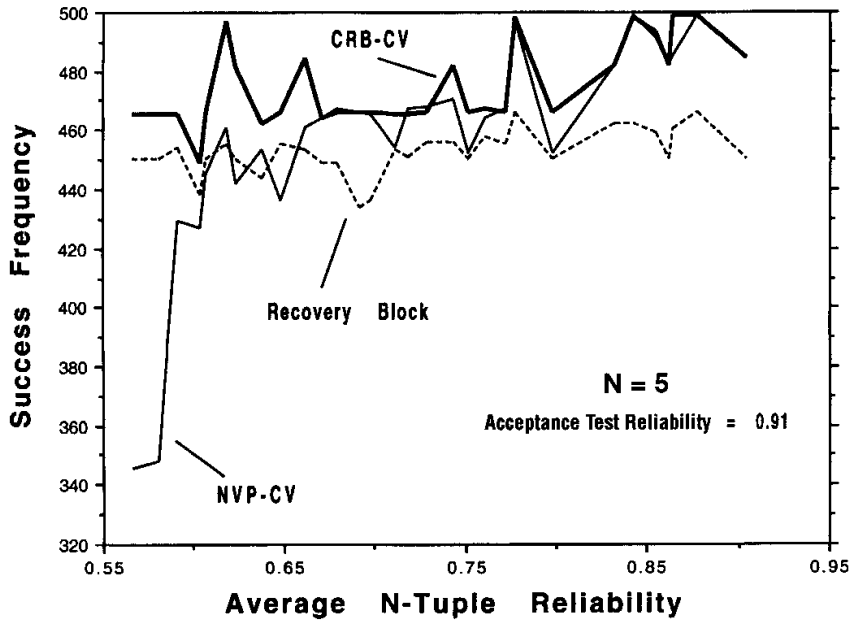


Figure 14.15 Consensus recovery block with consensus voting compared with N-version programming with consensus voting and recovery block.

when the acceptance test reliability is low, or when acceptance test and program failures are identical and wrong.

Similarly, consensus recovery block with consensus voting is usually more reliable than consensus recovery block with majority voting. However, if the number of agreeing versions is less than the majority, the reverse may be true. For instance, if there is no majority, then

majority voting will fail and the decision will pass to the acceptance test (which may succeed), while consensus voting will vote, and, if the plurality is incorrect because of identical-and-wrong answers, consensus voting may return an incorrect answer.

Both events described in the previous two paragraphs have been observed [Vouk93a]. In our experience, neither event is very frequent. A more general conclusion is that the consensus recovery block strategy appears to be quite robust in the presence of high interversion correlation, and that the behavior is in good agreement with analytical considerations based on models that make the assumption of failure independence. Of course, the exact behavior of a particular system is more difficult to predict, since correlation effects are not part of the models.

An advantage of consensus recovery block with majority voting is that the algorithm is far more stable, and is at least as reliable as N-version programming with majority voting. However, the advantage of using a more sophisticated voting strategy such as consensus voting may be marginal in high-correlation situations where the acceptance test is of poor quality. In addition, consensus recovery block will perform poorly in all situations where the voter is likely to select a set of identical-and-wrong responses as the correct answer (e.g., a binary output space). Instead, we could either use a different mechanism, such as the acceptance voting algorithm discussed below, or an even more complex hybrid mechanism that would run consensus recovery block and acceptance voting in parallel, and adjudicate series-averaged responses from the two [BelJ90, Atha89]. A general disadvantage of all hybrid strategies is an increased complexity of the fault-tolerance mechanism, although this does not necessarily imply an increase in costs [McAl91].

14.7.5 Acceptance voting

Acceptance voting is very dependent on the reliability of the acceptance test (AT). In some situations AV performs better than CRB, or any other voting-based approach. For example, AV reliability performance can be superior when there is a large probability that the CRB voter would return a wrong answer, and at the same time the AT is sufficiently reliable so that it can eliminate most of the incorrect responses before the voting. This may happen when effective output space is small (that is, voter decision space is small; see Prob. 14.6*b*). If AT is sufficiently reliable, AV can perform better than RB. In general, however, AV systems will be less reliable than CRB systems. Figure 14.16 illustrates experimental results for $N = 3$.

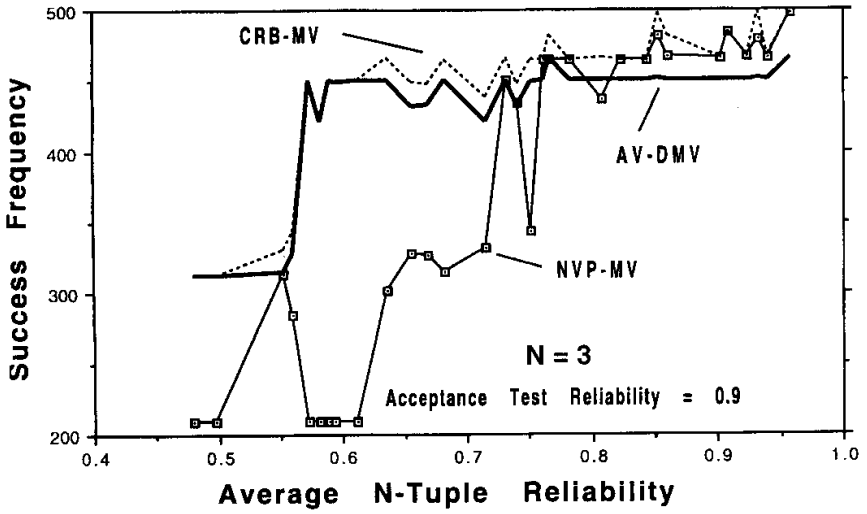


Figure 14.16 A comparison of CRB and AV, $N = 3$.

14.8 Development and Testing of Multiversion Fault-Tolerant Software

As we have seen, fault-tolerant software mechanisms often rely on the functional redundancy of components. This means that significant intercomponent dependence of failures may seriously endanger the effectiveness of these mechanisms. Hence, it is important to detect and eliminate faults that can cause correlated failures as early as possible. Unfortunately, current software production approaches are generally not geared toward early detection and classification of faults that can cause coincident and correlated software failures; neither is there a clear understanding of how this class of errors is generated. The FTS development issues are discussed in some detail in, for example, [Cris85, Kell86, Vouk86b, Voge87a, Kell88, Vouk88, Lapr90a, Lyu92b, Lyu93a].

Appropriate specification, design, coding, and testing approaches are essential, and in this context of special interest is practical use of formal methods, prototyping, reliability growth modeling, and methods for early detection of coincident failures.

A successful FTS development strategy for multiversion software should ensure the following:

- A small probability that the components participating in failure detection and correction decisions fail coincidentally. This includes independent as well as correlated coincident failures.
- High individual component reliability.
- High reliability of adjudication algorithms and their implementations, if adjudication is used, as well as low probability that the adjudication algorithm failures are related to those of the components.

- High accuracy and reliability of acceptance tests, if acceptance testing is used, as well as low probability that the acceptance test failures are related to those of the components.

14.8.1 Requirements and design

Two areas where FTS development must differ from single-component development, and where special attention should be focused, are software requirements and design. These phases are the potential source of a large number of correlated errors [Vouk90a].

Of course, we should use independent and isolated teams for development of multiple software versions or variants. But equally important is that the requirements are specified and analyzed using formal methods suitable for the problem being considered [IEEE94]. Risk analysis should be part of the approach (e.g., [Boeh86, Fran88]). Specification documents must be debugged and stabilized prior to being used for the development of the components. This can be achieved by developing prototypes (pilot code) of the final code. However, one must be wary of excessive coupling of the detailed prototype solutions with the specifications. This could eliminate beneficial fault-randomizing diversity usually provided by the detailed software design and implementation solutions.

Two decisions need to be made relatively early in the process: how many versions will be developed and which fault-tolerance strategy to use. Apart from economic considerations (see Sec. 14.8.3) the key questions in both cases are (1) how many faults need to be tolerated and (2) what level of fault-tolerance is expected.

In general, the larger the number of versions, the more faults can be tolerated. For example, two components are usually not capable of correcting for a failure of one of them. On the other hand, three versions tolerate one fault, four versions can tolerate up to two faults, etc. The real problem is whether or not an increase in the number of versions affects the number of correlated faults and its impact on the effectiveness of the selected fault-tolerance algorithm [Lapr90a].

The size of components is also an issue. A decomposition of a software problem into smaller components helps in making more precise decisions and effects better error control, but it adds processing overhead and also reduces diversity through a larger number of decision points, which require at least some amount of execution synchronization [Lapr90a, Vouk90b]. The choice of the fault-tolerance technique will be driven by trade-off analysis (modeling) of (1) the reliability gains, (2) performance requirements, and (3) cost, resource, and schedule constraints [Lapr90a].

During all phases of the development of multiversion software, it is necessary to follow a carefully designed and enforced protocol for prob-

lem reporting and resolution. Such a protocol must have provisions for ensuring independence of the development efforts and avoidance of more common sources of correlated faults—for example, communication errors and common knowledge gaps [Vouk90a] or casual exchanges of information among development teams [Lyu93].

14.8.2 Verification, validation, and testing

Verification, validation, and testing should be formalized and should strive to provide evidence of diversity, as well as evidence that individual components have high reliability and that there are no correlated failures either among the components or between the components and the decision algorithms (e.g., voting, acceptance test). With a process that includes careful inspection and testing of specifications, designs, and code, fault-tolerant software becomes a viable option for increasing reliability of a software-based system [Thev91].

A testing technique that, under the right conditions, may provide some help is back-to-back testing [Sagl86, Vouk90c, Lapr90a]. Back-to-back testing technique involves pairwise comparison of the responses from all functionally equivalent components developed for the FTS system. Whenever a difference is observed among responses, the problem is thoroughly investigated for all test cases where even one component answer differs. Of course, IAW responses are still a problem and need to be addressed separately through inspections and testing [Vouk90a]. However, excessive reliance on back-to-back testing can be counterproductive and can result in an overestimation of the component and system reliability [Bril87, Vouk90c].

As an illustration of the error-detecting power of back-to-back testing, consider an approximate bound on the effectiveness of this testing strategy assuming negligible intercomponent failure correlation. It is reasonable to assume that in practice an attempt will be made not to release software versions with any known faults, and that the versions will be tested to approximately the same level of reliability. Therefore we shall assume that all components have failure probability equal to f. The probability that n independent versions fail simultaneously on a test case is f^n. If the output space is binary, back-to-back testing will not detect a failure when all components fail, i.e., all answers are identical and wrong.

Hence, the probability that a failure will be detected is

$$P\{\text{failure is detected by back-to-back testing}\} \geq 1 - f^n - (1-f)^n \tag{14.27}$$

If we measure the efficiency of the processes through the probability that a failure is detected by a test case, we see that it depends on the program failure probability f and the number of versions involved.

The process of adding more components in order to increase failure-detection efficiency is one of diminishing returns under the assumption of independent failures. Let us assume that the probability of IAW responses is zero. Then, ideally, the fractional gain in failures detected by increasing n-tuple size from $n-1$ to n is

$$F(n) = \frac{1-(1-f)^n}{1-(1-f)^{n-1}} \tag{14.28}$$

It can be shown that for small f this fraction reduces to $F(N) = N/(N-1)$, that is, to the series 2/1, 3/2, 4/3, . . . ,1. This is illustrated in Fig. 14.17.

The fraction is plotted against the size of the larger tuple. Each point compares the efficiency of a k-tuple with that of the tuple one component smaller. We see that the incremental contribution to the efficiency of failure detection is considerably reduced for N-tuple sizes larger than 4 or 5. The line marked with $\hat{f} = 0.113$ shows experimental data obtained from a large multiversion experiment [Kell88, Vouk90c]. Note that the theoretical calculations are based on an interaction model that assumes failure independence, while in the experiment the interversion failure correlation was in excess of what would be expected if the failures were independent.

14.8.3 Cost of fault-tolerant software

Cost-effectiveness of fault-tolerant software is an open issue, although there is evidence that in some cases the approach is cost-effective.

For example, the experience of the Ericsson company is that diversity is cost-effective [Hage87]. Ericsson found that the costs of develop-

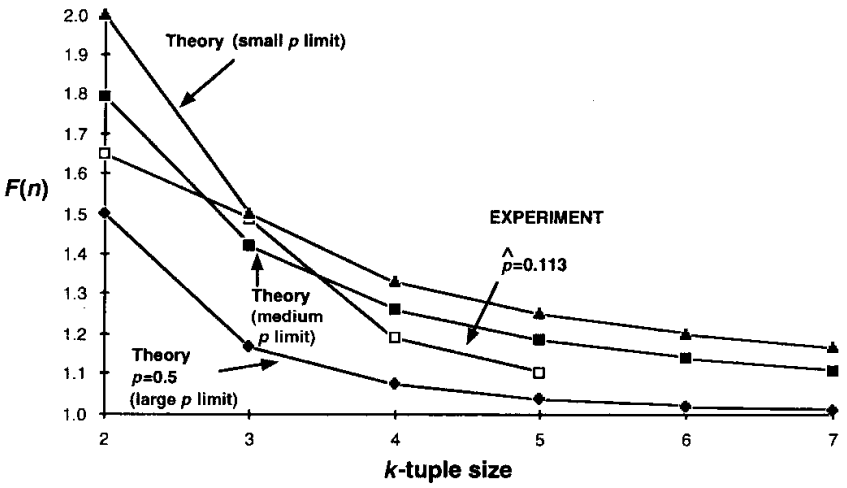


Figure 14.17 Illustration of the diminishing returns from successively larger k-tuples. The theoretical calculations are based on a failure model that assumes failure independence. Experimental data exhibited some positive failure correlation.

ing two functionally equivalent software versions is not double the cost of a single one for at least two reasons. First, not all parts of the system are critical, and therefore only the critical parts require additional work. Second, 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. There is also evidence that back-to-back testing is effective in discovering faults not detected by other testing techniques, and that it improves the quality of testing. There are other examples.

Panzl [Panz81] has found that dual-program development (with back-to-back testing) increased the initial development cost by 77 percent (not factor 2), but reduced the number of residual errors from 69 to 2.

[Voge87c] described several experiments related to the use of diverse programming and comparison testing. In one experiment 14 percent of the total number of detected faults were detected by back-to-back testing of three code versions *after* extensive individual testing of the versions was completed. It is also noted that dual programming was good not only for detection of implementation faults, but for detection of specification faults.

In the PODS experiment [Bish86, Bish88], again *after* extensive application of more traditional techniques, nine faults were detected by back-to-back testing, three of which were classified by the authors as "fail-danger" or critical faults. The cost of traditional testing was about 0.4 person-hours per test case, while the cost of back-to-back testing was about 0.4 person-hours per 250 test cases. It is also found that the cost of single-version development compared with threefold diversity (without back-to-back testing costs) was about 997 versus 2258 person-hours, i.e., an increase of about 126 percent, not a factor of 3.

Similarly, in [Shim88] authors report that 107 faults discovered by back-to-back testing were *not* detected by any other testing technique they used. However, they also note that the other testing found about 150 faults *not* detected by back-to-back testing they applied *after* the conventional testing.

Laprie et al. [Lapr90a] have analyzed the cost of a number of FTS approaches. They consider a number of sources of increased costs, such as the overhead due to decision-point synchronization, failure detection, and adjudication. They also consider a number of factors that reduce the cost per version, such as back-to-back testing, common test-harness development, and common test suites. Their findings are that in a multiversion setting a typical component cost is about 75 to 80 percent of single-version costs. This supports empirical results mentioned in the previous paragraphs.

The cost of FTS has been modeled by a number of researchers. The cost models are closely tied to reliability modeling and the underlying assumptions. The actual models are beyond the scope of this book, and

the interested reader is directed to [Bhar81, McAl85, Sagl86, Vouk90c, Lapr90a, McAl91, Lyu95a] and references therein.

14.9 Summary

A way of handling unknown and unpredictable software failures (faults) is through fault tolerance. In this chapter we introduced some techniques that can be used to develop fault-tolerant software. We focused our attention on methods that are based on redundancy through functionally equivalent software components.

The basic *idea* is that the required functionality is provided through a set of functionally equivalent software modules developed by diverse and independent production teams. In operation, all modules are given the same inputs. If one module fails, another can provide the correct answer. The underlying *assumption* is that it is possible to distinguish between incorrect and correct answers and continue system operation using the correct output. The basic *problem* is that it is conceivable that all modules will harbor the same fault. In operation, this may prevent distinction between incorrect and correct responses and a critical system failure may occur. The frequency of occurrence of such events in real-life software is still an open issue.

We have described some common and some advanced FTS techniques. For example, consensus voting is a generalization of N-version programming with majority voting. It provides adaptation of the voting strategy to varying component reliability, failure correlation, and output space characteristics. Since failure correlation among versions effectively changes the cardinality of the space in which voters make decisions, consensus voting is usually preferable to simple majority voting in any fault-tolerant system since it may provide some degree of protection from correlated failures.

Consensus recovery block is a hybrid technique that usually outperforms N-version programming. It also competes very successfully with recovery block in situations where the acceptance test is not of the highest quality. Consensus recovery block is surprisingly robust even in the presence of failure correlation. Acceptance voting is, under special circumstances, more reliable than both consensus recovery block and recovery block. However, in general, acceptance voting offers lower reliability than the other techniques.

We briefly discussed time-dependent issues and the need for such modeling, and FTS development and cost issues.

Problems

14.1 *a.* What is forward recovery and what is backward recovery? Give an example of each approach.

b. Using papers and books mentioned in Sec. 14.3, write a two- to three-page paper describing how error coding (e.g., convolutional coding) can provide information for fault tolerance (give at least one explicit example).

14.2 *a.* What is the difference between an acceptance test and an external consistency check? Construct a valid acceptance test for an algorithm that inverts matrices (include a list of assumptions and limitations).

b. Using papers and books mentioned in Sec. 14.3, write a two- to three-page paper describing how mathematical consistency checking and automatic verification of numerical computations can provide fault tolerance (give at least one explicit example).

c. Using papers and books mentioned in Sec. 14.3, write a two- to three-page paper describing how data diversity can provide fault tolerance (give at least one explicit example).

14.3 *a.* Use your calculator to compute the expression given in Eq. (14.5), and then repeat the calculation by hand. Explain, step by step, how the accuracy loss occurs.

b. Consider a floating-point system with base 10 and 5-digit arithmetic (that is, a mantissa of length 5 digits), the usual double-precision multiplication (10-digit arithmetic in this case), and rounding after every floating-point operation. You are given the following two numbers:

$$x = 0.10005 \times 10^5 \qquad \text{and } y = 0.99973 \times 10^4$$

Compute by hand the difference $x - y$.

c. Now assume that x and y are, in fact, the result of two previous multiplications and that the unrounded products that yield x and y are

$$x = \text{x1} \times \text{x2} = 0.1000548241 \times 10^5$$

$$y = \text{y1} \times \text{y2} = 0.9997342213 \times 10^4$$

Normalize and round to five places and then compute $x1 \times x2 - y1 \times y2$. The result should be 0.81402×10^1. It differs in every digit from the result obtained in item *b*.

Which is the correct result? Explain the anomaly.

14.4 You are given a recovery block system composed of N (diverse) versions of equal reliability and a perfect acceptance test. Derive the equation that describes its reliability.

14.5 Read Sec. 14.3.5 and consider the following. The following table shows responses of three programs to TOL = 0.1. The correct or "golden" value is 3.5 ± 0.1. The first column identifies the comparison event, the second one the actual event, and the last one the multiversion event. Although the numbers given in this table are hypothetical, all events have been observed in multiversion experiments described in [Kell88, Vouk90a].

			Events (TOL = 0.1, correct response = 3.5 ± 0.1)		
p_1	p_2	p_3	Comparison	Actual	Multiversion
3.4	3.5	3.4	Agreement	NO_FAILURE	ALL_CORRECT
3.5	3.5	3.5	Agreement	NO_FAILURE	ALL_CORRECT
3.4	3.5	3.6	Conflict	NO_FAILURE	FALSE_ALARM
3.4	3.5	3.7	Conflict	1_FAILURE	DETECTED_FAILURE
3.3	3.3	3.7	Conflict	3_FAILURE	DETECTED_FAILURE
3.3	3.3	3.3	Agreement	3_FAILURE	UN_DETECTED_FAILURE
3.3	3.3	3.2	Agreement	3_FAILURE	UN_DETECTED_FAILURE
3.3	3.4	3.4	Agreement	1_FAILURE	UN_DETECTED_FAILURE

An increase in the tolerance from 0.1 to 0.4 yields AGREEMENT for all comparisons. This eliminates the FALSE_ALARM event shown in the table, but also produces two new UN_DETECTED_FAILURE events. On the other hand, had the tolerance been made very small, most comparisons would have resulted in a CONFLICT increasing the incidence of FALSE_ALARM events, but also eliminating all but one UN_DETECTED_FAILURE event.

a. What is the difference between AGREEMENT and NO_FAILURE comparison events?

b. Construct a table comparable to the above given that in addition to version p_1, p_2, and p_3 we have version p_4 column with values as follows (3.4, 3.3, 3.5, 3.5, 3.5, 3.5, 3.3, 3.3).

c. Add majority voter and consensus voter columns (assuming perfect voters) to your table, and put in results of the decisions by these voters (use ACCEPT and REJECT as the two alternatives).

14.6 *a.* What is the minimum number of versions, and what is the smallest output space cardinality (assuming unique correct state) for which consensus voting makes sense (give an example).

b. Average conditional voter decision space (CD space) is defined as the average size of the space (i.e., the number of available unique answers) in which the voter makes decisions given that at least one of the versions has failed. We use CD space to focus on the behavior of the voters when failures *are* present. Of course, the maximum voter decision space for a single test case is N.

Add a column to the table generated in Prob. 14.5*b* that you will call "Decision space." The column should contain the number of *actual* answer categories (determined by tolerance) which are presented in each case to CV to make a decision.

14.7 Assume that version p_4 acts as an acceptance test, and that p_1, p_2, and p_3 form a voting 3-tuple. Repeat Prob. 14.5 using CRB3 and AV3 strategies.

14.8 *a.* Find the variance in the estimate for $\hat{f}_i$ (Eq. (14.16)).

b. What is the variance for the estimate for $\hat{r}_i$?

14.9 Let all versions in an NVP system have the same reliability. Let the voter be perfect. Then NVP with majority voting increases reliability over a single version only if the reliability of the versions is larger than 0.5.

a. Prove the above statement for $N = 3$.

b. Prove the above statement for any N.

14.10 Suppose N modules fail independently and have the same probability of failure f. Find the probability that exactly M modules of the N fail on a random input.

14.11 Compare the reliabilities of each of the redundant fault-tolerant techniques of Secs. 14.4 and 14.5 for two version systems. Assume that voting fails if the two versions do not agree. Assume the failure probabilities are identical for all components and that failures are independent.

14.12 Show that two randomly chosen programs p_1 and p_2 fail independently on a given input x with probability $\theta(x)^2$. What can you say about a specific program p failing on two randomly chosen inputs X_1 and X_2?

14.13 Construct an example of the function $v(p,x)$ over five programs and five inputs. Assume that the probability of selection of a program or an input is the same (1/5). Compute the functions $\theta(x)$ and $\phi(p)$. Find the probability that two randomly chosen programs will fail on a randomly chosen input.

14.14 Show that it is possible that random programs can fail on the same inputs yet still fail independently. Show that it is possible for random programs to fail on disjoint subsets of the input space yet fail independently.

14.15 Suppose we have independently developed N programs to solve a given problem. Exhaustive testing is impossible so we test the programs on a proper subset of the input space, repair them, and retest until they have no known errors. Is it possible that the corrected programs fail independently? Are the resulting programs randomly selected from the set of all programs?

14.16 Assume independence of module failures and analyze the reliability of the Airbus A310 system described in Sec. 14.5. Then assume equal module reliabilities and plot the system reliability as a function of module reliability.

14.17 Assume that specific software modules p_1 and p_2 have the same probability of failure. What can you say about the maximum difference $\{P[p_1 \text{ fails}, p_2 \text{ fails}] - P[p_1 \text{ fails}]\, P[p_2 \text{ fails}]\}$?

14.18 *a.* Derive the reliability polynomials for RB, CRB, and AV for the case that $r_1 = r_2 = r_3 = V = B = r$ in [0,1] and the faults are independent (do the derivation from first principles). Graph and compare reliability for each of the systems. Show that AV is inferior to all of the other systems in this case.

b. Perform sensitivity analysis for all three systems to independent faults in the decision nodes, that is, in the voter and acceptance tests.

For example, perform the analysis for the situation where the voter and acceptance test are perfect, where both have reliability of 0.99, 0.9, 0.85, 0.8, 0.7, etc.

c. Obtain and read reference [Kano93]. How do the results you obtained in item *b* compare with those obtained by Kanoun et al.? Explain.

14.19 *a.* Derive Eq. (14.23).

b. Use the Estimate II data in Table 14.1 to construct an independent-failures model. Give detail and explain your modeling work. Tabulate results.

c. The frequencies shown in Table 14.3 sum up to 500. Why is it that the sum of frequencies shown in Table 14.2 does not? Explain, using examples.

d. Compare the model in item *b* with experimental data given in Table 14.3 using the chi-squared test to comment on whether there is enough evidence to claim statistically significant departures between your model and the actual data for

Three or more failures

Majority

Twelve or more failures

e. Repeat the chi-squared test using Estimate I and Table 14.3 data. Are the test results different from those obtained in item *c*? Explain.

14.20 *a.* Write a two- to three-page paper discussing and justifying advantages and disadvantages of the following: (1) the desirable feature of a fault-tolerant software is *failure independence* among software variants and between software variants and adjudication algorithms; (2) the desirable feature of a fault-tolerant software is that software variants and adjudication algorithms have *disjoint failure sets.* Is either of the statements realistic?

b. Show mathematically that the assumption of module failure independence can be an overestimate or an underestimate of the probability of joint failure of two specific programs. What about two arbitrary programs?

14.21 Given three arbitrary functionally equivalent programs, the corresponding score functions, and density functions S and Q,

a. Derive an expression that states that at least two of the programs fail on an arbitrary input X.

b. Repeat item *a* but assume that the three programs are specific programs on an arbitrary input X.

c. How do the above expressions simplify if we assume the input space is finite and Q is a constant?

14.22 Suppose we are given a population of seven programs and three possible inputs. A table of the score functions for each program is given below: Assume that S and Q are constant.

v	x_1	x_2	x_3
p_1	1	0	0
p_2	0	1	0
p_3	0	0	1
p_4	0	1	1
p_5	1	0	1
p_6	1	1	0
p_7	1	1	1

a. Find $\theta(x_j)$ for each j.
b. Find $\phi(p_i)$ for each i.
c. Find $E(\Theta)$ and $E(\Theta^2)$ and variance squared of Θ.
d. Determine if two randomly chosen programs fail independently.
e. Compute the probability that p_i and p_j fail jointly.

14.23 Construct a table that has the following columns: *Method* (the name of the FTS method, e.g., RB, NSCP, NVP), *Failure detection mechanism* (e.g., by acceptance test, by comparison), *Failure tolerance mechanism* (e.g., rollback, reexecution, majority vote), *Number of functionally equivalent versions required to tolerate X independent failures* (e.g., $X + 1$ for RB).

a. Put into the table entries for RB, NVP-MV, NVP-CV, NSCP-MV, and NSCP-CV (MV—majority vote, CV—consensus vote).
b. Put into the table entries for CRB-MV, CRB-CV, AV-MV, and AV-CV.

14.24 Consider material in Sec. 14.8.2

a. Compare Eq. (14.27) with the RB reliability given in Eq. (14.20), and with NVP reliability given in Eqs. (14.23) and (14.24). What is the implication of Eq. (14.27), if any, on operational fault tolerance of these strategies? Is there an inconsistency? Explain.
b. Derive Eq. (14.28).
c. What is the implication of Eq. (14.28), if any, on the number of versions one chooses for a fault-tolerant system?

14.25 Show that if two programs fail independently on an input space of cardinality 2, then one program must always fail or never fail. Is this the case for an input space of cardinality 3?

14.26 *a.* What is back-to-back testing?
b. Read [Bril87, Vouk90c, Thev91] and write a two- to three-page discussion about the advantages and disadvantages of back-to-back testing as an aid in developing fault-tolerant software (include a discussion on why some authors believe that extended back-to-back testing may compromise N-version programming principles, and suggest some alternatives).