A Survey of Software Fault Tolerance Techniques

Jonathan M. Smith

Computer Science Department, Columbia University, New York, NY 10027

ABSTRACT

This report examines the state of the field of software fault tolerance.

Terminology, techniques for building reliable systems, and *fault tolerance* are discussed.

While a scientific consensus on the measurement of software reliability has not been reached, software systems are sufficiently pervasive that ‘‘software components’’ of larger systems must be reliable, since dependence is placed on them. Fault tolerant systems utilize redundant components to mitigate the effects of component failures, and thus create a system which is more reliable than a single component. This idea can be applied to software systems as well. Several techniques for designing fault tolerant software systems are discussed and assessed qualitatively, where ‘‘software fault’’ refers to what is more commonly known as a *bug.* The assumptions, relative merits, available experimental results, and implementation experience are discussed for each technique. This leads us to some conclusions about the state of the field.

1. Introduction

As computer systems become responsible for supporting increasing numbers of human activities, there is a corresponding increase in dependence on the machine’s correct function. The extremal points are *life-critical* systems [Leveson1986a], where the dependence on the system could determine whether a person lives or dies. Simple systems, e.g., a traffic light controller, can be constructed using only hardware components. More complex systems are constructed using programmable components, which require software (programs) in some form. These forms, for purposes of our discussion, can be, for example, Read-Only Memory (ROM), Erasable Programmable Read-Only Memory (EPROM), or some external device such as a magnetic disk. The software allows the actions of the general-purpose programmable hardware to be specified, thereby creating a system which behaves, in some sense, like a specialized piece of hardware.

Failures, which we will define precisely in a later section, can occur due to errors in the hardware (e.g., a short circuit) or errors in the software (e.g., using ’=’ instead of the intended ’==’ [Koenig1986a] in a C comparison). Hardware fault tolerance [Siewiorek1982a] is well-understood, to the point of being an engineering discipline. There are several reasons why this is so:

1. The physics of hardware components, such as silicon, are well understood;
2. The complexity of large hardware designs is several orders of magnitude less than large software systems;
3. Experience with built-in testing circuitry and data gathered by monitoring this circuitry has given engineers ‘‘rules of thumb’’ by which to design systems;
4. Given the costs associated with mass production, hardware engineers produce a carefully thought out specification, along with functional tests that can be applied in order to test units coming off the assembly line.

Software Fault Tolerance, the focus of our survey, is not yet an engineering discipline. In fact, in some respects , it retains an air of alchemy.

1.1. Survey Organization

We begin with a lexicon of terms in Section 2, for precision in the remaining discussion. Section 2 also outlines the role of fault tolerance in the design process, using a taxonomy developed by Laprie [Laprie1985a].

Section 3 provides a terse description of error recovery, and Section 4 discusses the nature of Software Faults, and also serves to relate software fault tolerance techniques to other methods for constructing reliable software systems, such as structured programming.

Section 5 discusses techniques which have been proposed for the construction of reliable software systems from less reliable components, in particular the Recovery Block, N-Version Programming, and Consensus Recovery Block. Assumptions, weaknesses, and experimental results of the techniques are presented as well. The section concludes with a qualitative discussion of the methods, and some quantitative data gathered by researchers.

Section 6 makes some observations, and concludes the survey.

2. Terminology

One of the useful factors which discriminates between science and art is a common terminology shared by workers in the field, with precise definitions. For example, in mathematics, *derivative* and *integral* are well-understood terms. Compare this to, e.g., wine-tasting, where terms such as *bouquet* describe the wine. From these examples, it's clear that a well-defined term can be used to *measure* and *compare,* and to share the results of these measurements. Researchers in Software Fault Tolerance have suffered from the lack of common definitions, recognized the state of their field, and attempted to improve it.

Morgan [Morgan1982a] points out where these problems with terminology have hindered researchers in fault-tolerant computing; efforts to remedy the problem are described. Lee and Morgan [Lee1982a] discuss the preliminary results of one such effort. In particular, they point out difficulties with definitions of the terms ''fault'', ''error'', ''failure'', ''specifications'', ''recovery'', ''system reliability'', ''hardware reliability'', and ''software reliability''. They suggest, as a start, that correctness and incorrectness be defined with respect to an Authoritative System Reference (ASR), which is in essence an ideal specification [Melliar-Smith1979a]. Disagreement with the ASR then constitutes a failure. The ASR can be inserted into a system in several ways; Figures 1 and 2 illustrate two of these:

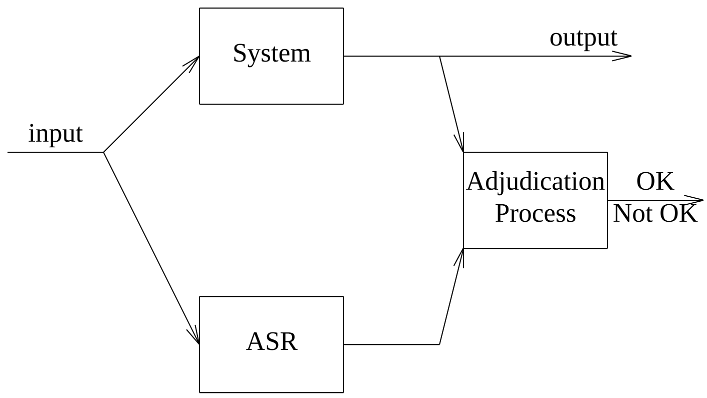

Figure 1: Lee and Morgan Figure 1a [Lee1982a, p. 35]

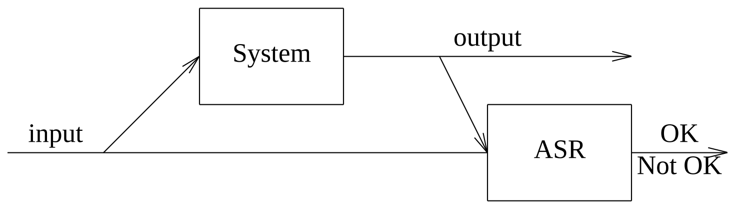

Figure 2: Lee and Morgan Figure 1b [Lee1982a, p. 35]

Lee and Morgan note that:

> "the definition does not imply that the failure is actually observed at the time it occurred, or that it is observed at all. The only requirement is that it could be observed by a strict application of the ASR." [Lee1982a, p. 35]

Laprie [Laprie1985a] also provides a discussion of terminology. In addition to bibliographic material, many examples for each term and a terminological taxonomy (reproduced in Figure 3) are given.

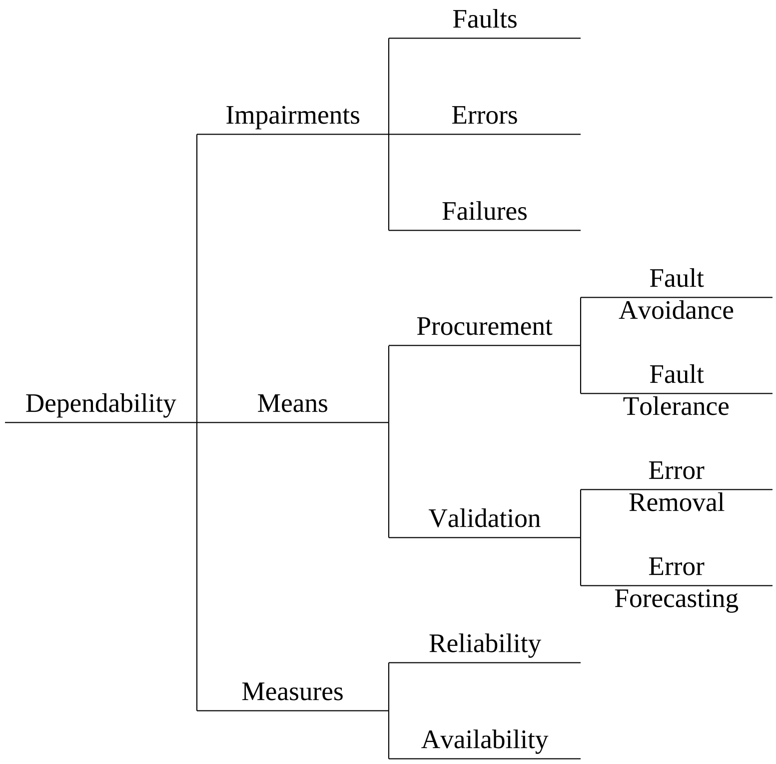

Figure 3: Laprie's Taxonomy

Our interest is in software systems which we can depend on to perform their function; thus ‘‘dependability’’ is the most abstract requirement we have. In order to satisfy this requirement, we have to know what makes systems ‘‘non-dependable’’: these are shown as ‘‘impairments’’, which are further discussed in the next section (2.1). The ‘‘means’’ are the methods with which we construct a dependable system, and the ‘‘measures’’ provide us with a criteria for evaluating our success. We shall concentrate on the ‘‘means’’ in this survey.

‘‘Procurement’’ is the methodology which is used to construct a dependable system, and ‘‘validation’’ the methodology used to ensure its dependability. An important distinction is that made between the two ‘‘Procurement’’ methods, ‘‘Fault Avoidance’’ and ‘‘Fault Tolerance’’. Highly reliable systems are constructed using combinations of two approaches, coupled with the information and feedback from ‘‘Error Forecasting’’ and ‘‘Error Removal’’. Fault prevention is the (optimistic) approach which uses forecasting information, and feedback from previous error removal, to eradicate potential faults from the system so that errors will not occur. **Fault avoidance** techniques attempt to avoid introducing additional faults. An example of fault avoidance from carpentry might be the use of the finest tools (to avoid broken tools) coupled with careful technique (e.g. use of a nail set to avoid damage to exposed wood surfaces). **Error removal** techniques attempt to discover and eliminate faults which were introduced; in the carpentry example, this would be an examination of the wood surface and filling holes with wood putty. Of course, the feedback from error removal can also be used in forming *fault tolerance* strategies.

Fault tolerance stems from the observation that it is rarely possible to carry fault prevention to its logical end, perfection. Fault tolerance recognizes that faults may exist, and fault tolerance strategies attempt to prevent faults from causing system failures. Four phases of fault tolerance are identified [Anderson1981a],

Error detection: The effects of the fault presumably manifest themselves in an error which results in an observable failure, which can be detected.

Damage assessment: A flagged error may only be the ‘‘tip of the iceberg’’; due to information propagation, much state information other than that directly associated with the erroneous state may be invalid. The extent of this damage should be discovered.

Error recovery: Error recovery attempts to transform the erroneous system state into an error free state. Two techniques are discussed below, in Section 3.

Fault treatment and continued service: While error recovery techniques may have brought the system back to an error-free state, further work may be required so that the system can continue to perform the activities in its specification.

2.1. Definitions

At a given level of decomposition, the system’s behavior can be described by a set of **external states** and a transition function between states; input data serve as stimuli for the transitions. If we separate the state information from the transition function, we are left with the following model for a system, illustrated in Figure 4:

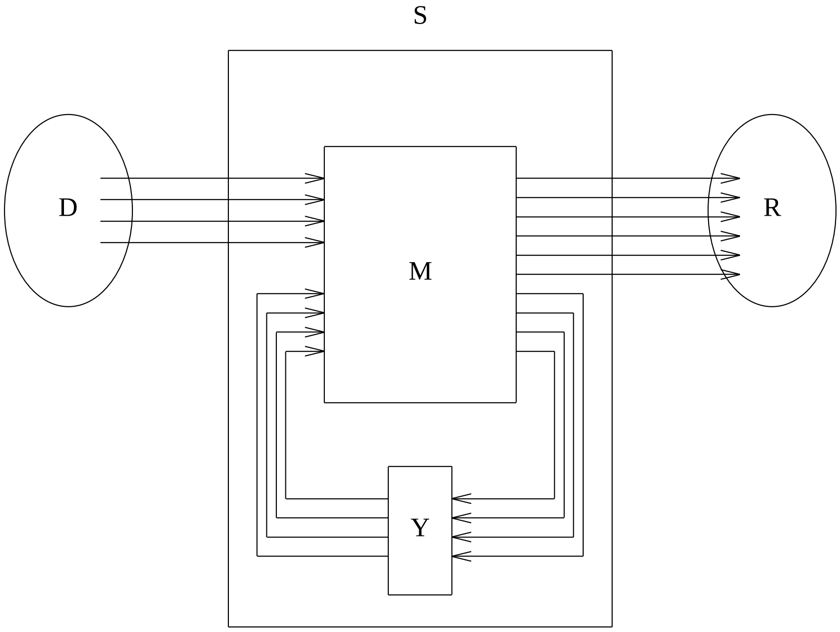

Figure 4: System as Finite State Machine with Feedback

This allows us to give formal definitions of terms:

— An *ideal* (intended) *system* **S** implements a *specified* mapping from a *Domain* **D** (input) to a *Range* **R** (output). Thus, **S** serves as the Authoritative System Reference mentioned earlier.

— **M** describes the control logic for the *state change* which takes place when $\vec{x} \in \mathbf{D}$ is presented to **S** to yield $\vec{z} \in \mathbf{R}$. Thus, **M** embodies all of the relevant transition functions to move between states; **Y** preserves all the relevant state information at any stage of the system’s operation. The value of $\vec{z}$ is obviously dependent on both $\vec{x}$ and the contents of **Y**.

— A version of **S** with an *error* is an $\mathbf{S}'$ with $\mathbf{Y}'$ such that $\mathbf{Y}' \neq \mathbf{Y}$. This inequality implies that **S** has at least one *erroneous state.* We note that the output may not be affected even if an erroneous state arises, i.e., $\vec{z} = \vec{z}'$ can still be true in the presence of an *error.* Thus, $\mathbf{S}'$ is an *incorrect* system because its state component $\mathbf{Y}'$ is not as specified. This may be due to externally-induced problems (e.g., bad memory) or to flaws in the control logic embodied in **M**.

— A *fault* stems from the existence within $\mathbf{S}'$ of a finite state machine $\mathbf{M}'$ such that $\mathbf{M} \neq \mathbf{M}'$. Thus, a *fault* is the difference between **M** and $\mathbf{M}'$ that caused the *error* (erroneous state) to arise.

— A *failure* in **S** is an $\mathbf{S}'$ such that (if we denote the output of **S**, $\vec{z}$, by a functional, **S**()) $\mathbf{S}'(\vec{x}) \neq \mathbf{S}(\vec{x})$ where $\vec{x} \in \mathbf{D}$.

$\mathbf{S}'(\vec{x})$ may not be an element of **R**. What this inequality between the outputs of **S** and $\mathbf{S}'$ means, less formally, is that the effect of the *fault* in $\mathbf{S}'$ has become *observable.*

— The *failure set* of an implementation $\mathbf{S}'$ is $\mathbf{F}' = \{\vec{x} \in \mathbf{D} : \mathbf{S}'(\vec{x}) \neq \mathbf{S}(\vec{x})\}$. Thus, by this definition, the failure set of **S** is $\emptyset$.

A slightly different set of definitions is given by Anderson and Lee [Anderson1982a], including some notions from Avizienis and Kelly [Avizienis1984a]. A **system** consists of a set of **components** which interact under the control of a **design.** A **component** is simply another system. A system having special characteristics is called the **design** - the design refers to that part of a system which supports and controls the interaction of the components[1]. Systems interact with **environments** at **interfaces** ; environments are clearly systems. This is illustrated in Figure 5:

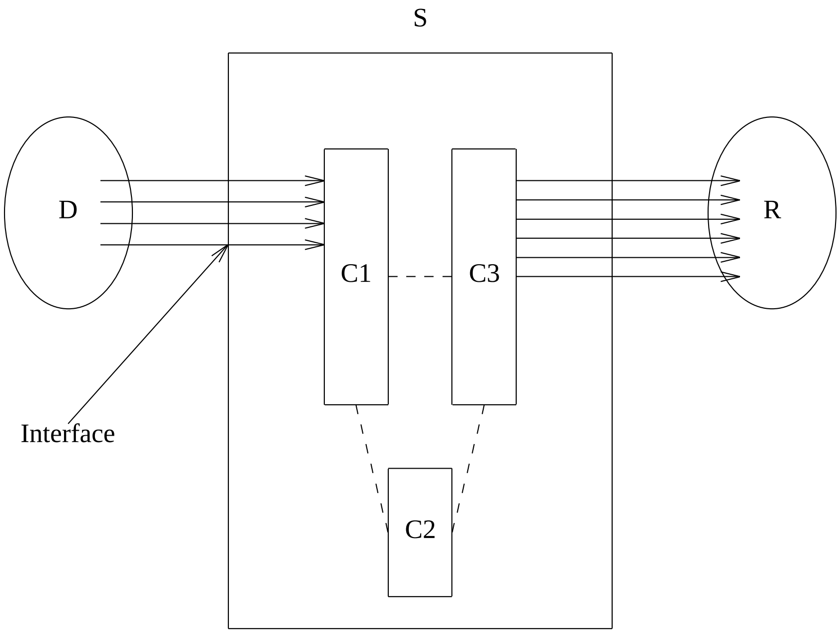

Figure 5: System constructed of components

In the figure,

— **S** is a **system.**

— **S** serves to map elements from an input domain, **D**, to an output range, **R**. Thus, **S** also defines a *mapping.*

— **C1, C2,** and **C3** are **components.**

— **C1, C2,** and **C3** are also **systems.**

— the dashed lines connecting the components are between **interfaces** in a **design;** the components shown thus comprise the **design** of **S**, when connected with the illustrated interfaces.

Note the recursive quality of these definitions; we can essentially continue applying them to components until the system is appropriately decomposed.

If a system is not atomic[2] it has an internal state comprised of the external states of its components.

A **specification** of the system's behavior allows judgements to be made about the **reliability** of the system; in particular, it allows one to distinguish between undesirable (behavior that is unwanted, but within the bounds set by the specification) and unreliable (not within the bounds set by the specification). An **authoritative specification** is one which can be applied as a test of the acceptability of the system. A **failure** of the system then can be defined as the initial deviation of the system from such an authoritative specification; the reliability of a system is specified as a function of failures and time.

---

[1] This definition allows the notion of a ''design fault'' to be clarified; the design is the logical ''glue'' used to combine components.

[2] ''Atomic'', in this setting of recursive decompositions, means that the internal structure of the system is not of interest.

An **error** occurs when a fault (internal to a component or design) manifests itself in the form of a defective system state (detectable by means of affected external behavior). An error in a component or in the design of a system is a **fault;** this distinction is made with respect to the level of the system's decomposition; thus the fault in the component (or design) level causes the error at the system level.

The important distinction between the two sets of definitions illustrated by Figures 4 and 5 is the system model. While the recursive decomposition of a system shown in Figure 5 is quite elegant and general, it is not as useful as the more formal single-level model in achieving precise definitions, which in turn can be used in making quantitative statements. The single-level model has the additional advantage that when reasoning about any system's behavior, only one level of each component is of interest at any given analysis point, and thus the ''flat'' decomposition as a finite state machine is entirely adequate, and seems most appropriate.

3. Error Recovery

Assuming that the error detection mechanism has discovered an error, we can either indicate the error and fail, or attempt to recover from the error and any effects it may have. The essential goal of any error recovery strategy is that the system be in a correct state, with respect to the specified behavior. We must assume that at some time the system was in a correct state, and that some later time, it was no longer in a correct state. At the time which the system moved from a correct state to an incorrect state, an *error* occurred. Two approaches exist to attaining a correct state, forward error recovery, and backward error recovery. These are discussed in the following two subsections.

3.1. Forward Error Recovery

Forward error recovery is so named because the scheme moves forward along the time line while attempting to recover. The basic idea is that once an error is discovered, steps are taken to move the system from its current incorrect state into a state which is correct. In our state machine formalism, the $\mathbf{Y}'$ must be transformed through some means into $\mathbf{Y}$, in spite of the existence of $\mathbf{M}'$ or externally-induced errors. This approach has the major flaw of requiring designers to be aware of the possible failure modes of the system in advance, as the logic necessary to move the system from an incorrect state to a correct state must be constructed into the system.

3.2. Backward Error Recovery

Backward error recovery relies on the fact that the system was in a correct state at some time in the past. This is used in the following way. At a time when the system is known to be in a correct state, a copy of the system state information is made; call this copy the ''backup''. When the system is determined to be in an incorrect state, the backup copy of the system is used to restore the system, thus achieving a correct state by moving ''backward'' in time. In our state machine model, the error, as instantiated by $\mathbf{Y}'$ is removed by restoring a previous $\mathbf{Y}$. This scheme (as described) has a major flaw when faced with recurring errors; the problem that caused the rollback[3] may not have gone away, in which case the system will fail again. This can be dealt with (to some degree) by using an alternative method of moving forward after the system state has been restored.

4. Software Faults

Hardware faults can be the result of physical degradation or they can result from poor design. Software however can only have faults due to mistakes in design and implementation; it does not suffer from functional changes due to external interaction. We can state the nature of software faults in the terminology developed in the previous sections: A ''software fault'' (bug) is a *fault* in a *system* **S** which is implemented in software.

There are two approaches to constructing (i.e., Laprie's ''procurement'') highly reliable software systems, *fault avoidance* and *fault tolerance.*

[3] ''Rollback'' is a colloquial term for restoring state derived from ''rolling back the clock''.

4.1. Fault Avoidance

The basic notion of fault avoidance is that the systems should be as fault-free as they can be made. For example, in software, *failures,* and *faults* causing errors, can be reduced or removed by:

- Verification (proofs of correctness) [Floyd1967a, Hoare1969a, Dijkstra1976a]
- Precise specifications combined with testing. Prather [Prather1983a] provides an excellent overview of program testing, and includes an extensive bibliography.
- Structured programming techniques [Dahl1972a] with which we combine the numerous design techniques such as Parnas' information hiding [Parnas(null)a], Ross' SADT, *et cetera.*

However, experience has shown that these techniques are insufficient, i.e. ''bugs'' remain. Intuitively, it should be clear that testing does not prove that no bugs remain, unless both the specification is complete in every detail, and the testing exhausts all the possibilities specified. To quote Dijkstra [Dijkstra1972a] :

> ''Program testing can be used to show the presence of bugs, but never to show their absence''

Proofs of program correctness are an elegant methodology for producing correct programs, but they suffer from the same weaknesses of any mathematical proof, that is:

- What has been proved is what has been specified. For example, it can be shown that a sorting program produces a lexicographically ordered set of strings as an output, given that a set of strings has been provided as input. What is implied, not proven, and necessary is that the output set be a permutation of the input set; hence, while my ''proof'' would be correct, it does not address the problem.
- The proof can be believed. As the complexity of mathematical proofs increases, the complete proof becomes incomprehensible, with 200 page proofs not uncommon. With automated logic, this is less of a problem, but another problem is associated with program size: as the program becomes larger, so does the complexity of proving it correct, so that only small programs can be handled with these techniques. To be fair, the techniques are intuitively attractive, and once the process of using the automated verification's error detection capabilities became familiar, confidence in the techniques would improve.

Structured programming techniques are disciplines, which when applied can lead to faster development times and fewer errors [Brooks1975a], yet they do not guarantee that specifications are followed, nor that the modular decomposition done is the correct one for the problem at hand.

Gerhart and Yelowitz [Gerhart1976a] provide a host of examples where published applications of these techniques are in error; they also attempt to describe some of the general problems which cause the errors.

One design technique which has been successful is the use of High Level Languages. This use is predicated on the observation that, on average, a programmer produces a constant number of fully-debugged source lines per day, and *the number stays the same independent of language*. Thus, for a given task and time frame, error-free code will be produced faster using a higher level language. In addition, the tools available are very helpful in removing errors early in the process, e.g. type-checking compilers and support systems [Strom1983a]. However, the task size may be too large, or the time available too small for these methods to make the deliverable error-free, in spite of the quality of these tools.

Given that the approach of fault avoidance cannot yet provide us with fault-free software, the question must be asked: is there a way to achieve high reliability in spite of these problems?

4.2. Software Reliability Assessment

One should of course have a firm grasp of what ''reliability'' is in the context of software, and how to measure it. Littlewood [Littlewood1979a] addresses this concern in a short discourse on measurement techniques applicable to software. Several software reliability models and metrics have been suggested [Musa1975a, Musa1979a, Littlewood1979b, Goel1979a] Musa, Iannino, and Okumoto [Musa1987a] provide a detailed reference on their approach to software reliability; the survey paper by Ramamoorthy and Bastani [Ramamoorthy1982a] discusses a wide variety of models and issues. From this evidence, it seems that predicting and estimating reliability of software are difficult tasks indeed, and of course new methods make data gathered using, e.g. assembly language, somewhat obsolete. This, unfortunately, makes science hard. As Feynmann [Feynman1963a] points out, ''...the sole test of the validity of any idea is experiment'', and without metrics and gathered data, this is difficult.

However, in spite of this lack of experimental data, general techniques have been proposed to increase the reliability of software. As we point out in the conclusion to the survey, it seems prudent to put effort into the design phase rather than trying to remedy problems after the fact. However, there are situations where the ‘‘gold-plated’’ approach may not be applicable, and the techniques in the next section try to address such situations.

5. Software Fault Tolerance

Fault tolerance is a technique based on the somewhat pessimistic assumption (although *de facto* realistic) that we haven’t removed all the bugs. The basic idea behind all of fault tolerance is the use of *redundancy.* The idea of using redundancy to construct reliable systems from unreliable components was first described by Von Neumann [Neumann1956a] in 1956. *Redundancy* (multiple copies) is used to *detect faults* and *mask failures.* Avizienis and Kelly [Avizienis1984a] suggest that the different types of redundancy possible in a computation are:

- repetition
- replication (hardware)
- logic (software)

These are of course old ideas applied in a new setting; for example, the same methods are used in experimental science to gauge the reliability of results; thus science sets the criterion that experiments must be *repeatable* in order to be believed.

We will use the formal state machine model of Section 2 to discuss redundancy. The use of redundancy relies on *statistical independence*. In statistics [Robbins1975a], *independence* means that given two *events,* $\mathbf{A}$ and $\mathbf{B}$, from an *outcome space*, $\mathbf{I}$, $pr(\mathbf{A} \mid \mathbf{B}) = pr(\mathbf{A})$, and $pr(\mathbf{B} \mid \mathbf{A}) = pr(\mathbf{B})$. Thus, $pr(\mathbf{A} \; \mathbf{B}) = pr(\mathbf{A}) \cdot pr(\mathbf{B})$. Since *faults* are the events of interest in our discussion, if $\mathbf{S}_1$ and $\mathbf{S}_2$ behave independently with respect to $\mathbf{D}$, $pr(\vec{x} \in \mathbf{F}_1 \; \vec{x} \in \mathbf{F}_2) = pr(\vec{x} \in \mathbf{F}_1) \cdot pr(\vec{x} \in \mathbf{F}_2)$. A *fault* is common to systems $\mathbf{S}_1$ and $\mathbf{S}_2$ *iff* for $\vec{x} \in \mathbf{D}$, $\vec{x} \in \mathbf{F}_1$ and $\vec{x} \in \mathbf{F}_2$, where $\mathbf{F}_i$ is the subset of $\mathbf{D}$ constructed from the inputs on which $\mathbf{S}_i$ fails. What this means less formally is that the fault is common to both systems If we concern ourselves with only the $\vec{x}$ such that $\vec{x} \in \mathbf{F}_i$, we can see that the relation between $\mathbf{F}_i$ and $\bigcup_{i=1}^{N} \mathbf{F}_i$ (the set of inputs on which at least one of N systems, some $\mathbf{S}_i$ fails) is interesting. Analyzing this membership data, however, provides us much more important information, which is:

- The inputs $\vec{x}$ such that more than one version (some $\mathbf{S}_i$) fails at $\vec{x}$.
- The number of versions which fail at $\vec{x}$.

The number of versions which fail at $\vec{x}$ can be compactly described by using the notation

$$m(\vec{x}, N) = \sum_{i=1}^{N} e(\mathbf{F}_i, \vec{x})$$

for a given set of systems $\mathbf{S}_1, \ldots, \mathbf{S}_N$ where for set b and element a

$$e(b,a) = \begin{cases} 1 & a \in b \\ 0 & otherwise \end{cases}$$

serves as a set membership operator.

Thus, for a given $\vec{x}$, the relationship between the errors and the common data domain for three systems could be described as $m(\vec{x}, 3)$.

5.1. Independence Assumptions

We now discuss *coincident errors* in relation to the formal model of systems, faults, and errors which we presented earlier. Reasoning about experiments and events often assumes that the events are independent (although fervent gamblers might disagree). Then, the reproducibility of experiments implies that the result of the experiment (an event) was not a coincidence. The more times and places the experiment is performed, the greater faith we have in the results. Thus, redundant systems which fail independently of each other can provide a gain in reliability. This is important information if we seek to estimate reliability of the final system from observations we have made about its constituent components. Where reliability goals are set, this information can predict how many independent copies of software are necessary to meet the goal.

Knight and Leveson [Knight1986a, Knight1986b, Knight1985a] have carried out experimental work on the independence assumption of multiple software versions. Their results indicate that the assumption of independence with respect to the faults in the software is not upheld in practice. Eckhardt and Lee [Eckhardt1985a] provide a theoretical analysis of coincident errors and their effects on some software fault tolerance schemes. Other results are discussed later in the section on the "Consensus Recovery Block". However, even if coincident errors constitute a large enough fraction of the errors such that statistical independence cannot be assumed, if we can *combine* $\mathbf{S}_1$ and $\mathbf{S}_2$ and *decide* how to select which $\vec{z} \in \mathbf{R}$ to use, we may decrease the number of failures relative to the input set. Several techniques, discussed in the following sections, have been proposed as methods of combining systems and selecting results. Anderson [Anderson1986a] has proposed a framework which includes these combining and selection techniques. The combining mechanism combines the multiple software versions. The selecting mechanism selects a result from the combined multiple versions of software. These two mechanisms comprise a software fault tolerance scheme.

5.2. N-Version Programming

N-version Programming is a technique originated and advocated by Avizienis, et al. [Chen1978a, Avizienis1977a, Avizienis1985a]. Much research on the technique has been published, starting in the mid-1970s. N-version programming is conceptually similar to N-modular redundancy, a hardware reliability [Avizienis1978a, Siewiorek1982a] technique. The basic idea is that N versions of a software system, (where N is an odd positive-valued integer) are executed with $\vec{x} \in \mathbf{D}$. The *decision algorithm* used to resolve differences in $\vec{z}_{(i)} \in \mathbf{R}$ is *voting;* a vote[4] determines which $\vec{z}_{(i)} \in \mathbf{R}$ is chosen as *correct.* The simplest case in achieving agreement is a test for bitwise equality as it is independent of the application. The *decision algorithm* relies on *independence* because the probability that an $\vec{x} \in \mathbf{D}$ is an element of more than $\frac{N-1}{2}$ of the N failure sets $\mathbf{F}_1, \mathbf{F}_2, \ldots, \mathbf{F}_N$ should decrease exponentially as N increases. N-version programming fails at $\vec{x} \in \mathbf{D}$ such that $m(\vec{x},N) > \frac{N-1}{2}$, where $m(\vec{x},N) = \sum_{i=1}^{N} e(\mathbf{F}_{(i)},\vec{x})$ is as described above. Thus, if *failures* do **not** occur *independently,* predictions of the reliability of N-version programming which assume independence will be false. The number of *coincident errors,* (m above), determines the set of $\vec{x} \in \mathbf{D}$ such that N-version programming will fail.

Some practical issues of constructing such systems are being addressed by the DEDIX [Avizienis1985b] system under construction at UCLA.

5.3. Recovery Block

The Recovery Block method was originated by Horning, Randell, and others [Randell1975a, Horning1974a] at the University of Newcastle upon Tyne. Others associated with that research group [Anderson1981a] have discussed recovery blocks and produced special-purpose hardware [Lee1980a] to support efficient implementation of the scheme. A compendium of this work has been compiled by Shrivastava [Shrivastava1985a].

The recovery block is a language construct for encapsulating a program segment which is to be performed reliably. Semantically, it behaves in a manner similar to block structuring in programming languages, in that a block has both private variables and access to global variables (those declared external to the block). The recovery block scheme defines a method for ensuring that the changes effected to external variables be done reliably. The scheme is conceptually similar to the ‘‘standby spare’’ technique used in constructing reliable hardware systems [Toy1983a, Hansen1983a]. N alternate methods of passing an *acceptance test* (instances) are provided and rank-ordered according to some criterion such as observed performance. The first such instance is referred to as the *primary.* Each instance is tested, and the first which passes the *acceptance test* provides the *result* of the *recovery block.* It is important to note that each *instance* is *guaranteed* to execute with the system in the same state as before the *recovery block* was entered. In Anderson’s framework, mentioned earlier, the *decision algorithm* is the acceptance test. *Assuming* that the *acceptance test* performs as intended, the Recovery Block method fails at $\vec{x} \in \mathbf{D}$ such that $m(\vec{x},N) > N-1$, i.e., $m(\vec{x},N) = N$. Figure 6 gives an example, which we’ll explain in the next section, of a Recovery Block designed to perform a numerical calculation.

---

[4] There are several ways that the vote could in fact be implemented. For example, a 2-out-of-N scheme could be used, where as soon as 2 respondents agree on a result, the result is determined to be valid. However, the rule is typically majority vote, as this intuitively provides the greatest protection against random errors. We will assume majority vote in the remainder of the discussion.

```
#define TOLERANCE (1.0e-5)
#define EQUAL(_a,_b) ((((_a)-(_b))/(_b)) < TOLERANCE)

double ft_sqrt( x )
double x;
{
   double y, newton(), bisection(), fail();

   ENSURE EQUAL( y*y, x )
   BY
      y = newton( x );
   ELSE_BY
      y = bisection( x );
   ELSE_ERROR
      y = fail();
   END

   return( y );
}
```

Figure 6: Simple Recovery Block example

5.3.1. Notation

The goal of the routine is to provide an output which is the square root of the numerical argument. The notation we have used in the example is that of the RB [Smith1987a] language of Smith and Maguire; the notation for specifying recovery blocks closely follows Randell [Randell1975a].

The **ENSURE** keyword indicates that what follows is to be used as the acceptance test for this recovery block. In this case, we have defined a macro EQUAL which defines equality in terms of a relative error measure to make the example more realistic.

The **BY** keyword ends the specification of the acceptance test and denotes the beginning of the primary alternate. **ELSE_BY** is used to specify further alternates; the **ELSE_ERROR** keyword specifies arbitrary code to be executed upon failure of the set of alternates to produce an acceptable answer. The **END** keyword terminates the recovery block.

Note that the acceptance test is specific to this application; by its nature, this will almost always be the case. Hecht [Hecht1979a] provides a detailed discussion of the forms such acceptance tests might take. Scott, Gault, and McAllister [Scott1983a] and Scott, et al [Scott1984a]. have shown that acceptance test failures can be tolerated within a certain range, in particular failure rates f , $0.0 \leq f \leq 0.25$. Scott's thesis [Scott1983b] proposes a synthetic software fault tolerance method, which is discussed in the next section.

5.4. Consensus Recovery Block

Scott [Scott1983b] observed the following in an analysis of the N-Version Programming and Recovery Block schemes for software fault tolerance:

1.) Recovery Blocks: The major difficulty with the Recovery Block scheme is the acceptance test; analysis shows that it is the most crucial component of the scheme if reliability is to be increased over that of the primary alternate[5]. In fact, Scott points out that highly reliable primary versions may succumb to imperfect acceptance tests which, for example, declare a correct result to be incorrect. Thus:

> ''Using this information, a software manager may conclude that if he can develop a very reliable program then there is no reason to consider fault-tolerance in the form of a Recovery Block.''

(One potential flaw with the analysis is that Scott considered only reliability. If we have a **very** reliable acceptance

[5] Three alternates were rank-ordered based on reliability, determined by testing. Thus, the primary alternate is also the most reliable. Another rank-ordering based on execution-time performance, or a product of reliability and performance, might have been chosen.

test, and two alternates, one 90 percent reliable and requiring 1 time unit to complete, and the other 99.9 percent reliable and requiring 10 time units to complete, we can get higher performance by putting the faster alternate first, for an average performance of 1.9 time units compared to an average performance of 9.99 time units for the other ordering.)

Given the importance of the acceptance test, the following seem to be major flaws:

- There are no guidelines for developing a test.
- There is no methodology for testing.
- The acceptance test, being written in software, is subject to design faults.
- There are no general guidelines for placement and implementation of Recovery Blocks in a software system.
- The acceptance could be expensive to compute.

2.) N-Version Programming: The major difficulty with the N-Version Programming scheme is the voting aspect of the design.

- Multiple correct outputs may occur.
- Exact equality may be required in cases where multiple versions have computed the same result to varying degrees of precision. Otherwise special purpose result comparison logic (which may contain faults) is necessary.

Given these observations, Scott concluded that a synthesis of the two systems might alleviate some of the problems of the Recovery Block scheme. This synthesis of the two schemes is called the ‘‘Consensus Recovery Block’’, and works as follows:

1.) N independently developed versions of a specified program are available.

2.) An acceptance test for the results of the program is developed.

3.) A voting procedure exists.

4.) The versions are rank-ordered based on a set of criteria, as in the Recovery Block Scheme.

5.) All versions execute and submit their outputs to the voting procedure.

6.) The first two of the N which agree are considered ‘‘correct’’ under the assumption of independence, and not subjected to the acceptance test.

7.) If no agreement (consensus) is reached, the alternates’ outputs are examined using the ordering; the first successful alternate is marked ‘‘correct’’; if none are successful, failure occurs.

The proposed advantage of the scheme is in avoiding the acceptance test as a decision mechanism in cases where there is agreement in 2-of-N of the voters; thus a relatively unreliable acceptance test may not have much impact on the Consensus Recovery Block’s output. Note also that like the N-Version Programming Scheme, there is no requirement for state recovery and the possibility for truly parallel activity exists.

In both the N-Version Programming scheme he studies, and the Consensus Recovery Block, Scott assumes that 2-of-N achieves a consensus. He contends that under an independence assumption, the probabilities are such that a majority scheme provides little incremental value. In any case, it makes the model somewhat easier to understand and analyze. With the independence assumption, Scott proves that the Consensus Recovery Block is more reliable than N-Version Programming or the Recovery Block scheme in almost all circumstances, and never less reliable.

5.5. Discussion

Much of the analysis of the N-Version Programming scheme and the Recovery Block scheme is provided in the previous section on the ‘‘Consensus Recovery Block’’ method. However, some other issues should be addressed. From the programmer’s point of view, N-Version Programming needs no extra logic beyond the program flow itself, unlike the exoalgorithmic ‘‘self-check’’ embodied in the acceptance test. This reduces the intellectual burden of using the method, and with this reduction in complexity comes a reduction in software faults.

However, it should be observed that in spite of its conceptual simplicity, a practical implementation of N-Version Programming is subject to many of the same implementation problems as a Recovery Block scheme; that is, the support mechanisms themselves may not be reliable. For example, there is the need for some support mechanism to spawn and collect results from the N versions. The synchronization mechanism may need timeouts, and the voting mechanism is also subject to design faults, although as support software, there is a strong incentive to apply verification and other techniques to ensure their correctness.

However, there remain points to be addressed. In particular, how well do the schemes perform in practice? Scott's experimental data [Scott1983b] provide a great deal of insight into the assumptions and behavior of these methodologies.

First, he concludes that the independence assumption is unwarranted[6] based on a statistical analysis of many independently-developed[7] versions of a program. His analysis of this situation is that independently-developed programs tend to fail on the same problems because the difficult cases remain difficult across versions. Thus, the independence or lack of it is as much a characteristic of the problem space as it is of the solution space.

Second, he concludes that the Recovery Block scheme provides an increase in reliability over the best single-version programs in each group of 3 programs in his experiment when the reliability of the acceptance test is greater than 0.75. This was true with varying component reliabilities.

Third, he concludes that N-Version Programming actually decreases the reliability of the programs in his experiment; that is, that the reliability of a 3-Version Programming scheme (3 is both 2-of-N and majority) is less than that of the most reliable version of the 3 used as components. This seems intuitive, as the N-version Programming scheme will fail whenever the best version is correct and the other two less reliable programs either disagree or agree on another output. Scott notes that the problem of multiple correct outputs particularly plagued the N-version programming scheme.

Finally, the Consensus Recovery Block does not perform as well the Recovery Block scheme, but is more effective than N-version Programming when the versions are not independent. The synthetic method offers an improvement in reliability over the single best program in a group of 3 programs when the acceptance test is sufficiently reliable, e.g., 0.90.

Thus, given the fact that software failures are not independent, it appears that the Recovery Block scheme provides the best protection against software faults. While Scott argues that acceptance tests are the main difficulty of the recovery block scheme when presenting the Consensus Recovery Block, he later states (when describing the construction of his experiments and the acceptance test for them):

''... It was determined that such a simplistic acceptance test could be programmed without error...''

6. Conclusions

We have focused our attention on *general* methods for software fault tolerance. Certain problems allow problem-specific fault-tolerance strategies to be used, e.g., where some redundant information such as an ''invariant'' is part of the problem definition. Lerner [Lerner1988a] has examined these techniques, and finds them promising, especially in massively parallel architectures. The more general algorithms, such as the ones we have discussed here, must function without regard to the problem characteristics. In this sense, N-Version Programming is the most general technique, since the writing of the Recovery Block's ''acceptance test'' requires some problem-specific redundancy, like the methods Lerner discusses.

Experimental data gathered to date indicates that the Recovery Block method is most effective, but the evidence is not sufficient to scientifically conclude its superiority. This is true for two reasons. First, as we mentioned in the body of the survey, the measurement of software reliability is immature. This is most in evidence when we try to *estimate* the reliability of a piece of software assuming some metric. Thus, it is hard to rigorously compare, and thus rank and improve, software reliability techniques.

Second, there has been insufficient experimental work done and reported. Existing experimental work can be divided into two types. These are time-domain and data-domain testing. Time-domain testing is typically used in very large software systems, where the concern is of the nature ''when is it going to break next?''. The management of the development process requires logging of failures, their times and natures. This seems most appropriate for the domains in which it has been applied, e.g. ABM software [Musa1987a]. Data-domain testing uses *a priori* knowledge of the input set to determine where the programs fail, and why. This seems most appropriate for the multiple-version software approach. Experiments such as those by Knight and Leveson, *et al.* should be performed on a greater variety of programs than the commonly-used missile-tracking example.

Unfortunately, these experiments are both expensive (as they require many copies of a software system to be written)

[6] And thus concurring with the Knight and Leveson results cited earlier.

[7] It is possible to criticize such an experiment on the basis of the programmers being students in the same class, with the same training. Then again, academic environments place constraints on sharing of information not posed by the real world, e.g., accusations of plagiarism.

and difficult, as they involve humans, so that the control must be done carefully in order to maintain the validity of the conclusions. The expense of multiple versions often means that classroom situations are used to gather experimental data. These data may not be representative of the program characteristics in situations where high-reliability software systems are needed. The economics of these experiments usually preclude much industrial participation, yet industrial (particularly defense industry) settings provide the most obvious applications of this technology.

As we set Software Fault Tolerance into the context of producing reliable systems, we also see that the production of reliable software can consist of several interrelated phases (see Figure 3). We have argued, in defense of software fault tolerance, that ‘‘Fault Avoidance’’ in the ‘‘Procurement’’ phase has not been entirely successful. Yet, there is the system design question of where to spend time, money, and labor to yield a reliable product.

The consensus, an engineering principle almost, is that the time should be spent in the design phase. An example can be drawn from automotive engineering. Cars are fairly complex systems, and thus in a sense are like our computer systems. An automobile is a particularly well-chosen example in that while people rarely duplicate components such as complete automobiles (can you imagine towing a spare car!), fault-tolerance techniques are applied where reliability is important. For example, the brakes of my current automobile are a dual hydraulic system. Each of the hydraulic systems controls three wheels; thus I am capable of braking even if one fails. Thus, engineering talent was spent on the design phase, and the design includes fault-tolerance.

In addition, we should note that while design effort is certainly spent on making brake systems reliable, there are few, if any, automobiles sold without an emergency brake!

7. Acknowledgments

Gerald Q. Maguire, Jr. participated in all phases of the writing of this survey, from suggesting material to critical review. Discussions with Yechiam Yemini and Calton Pu were helpful in deciding what material had relevance. Discussions with Nancy Leveson and Rob Strom led to a much better understanding of Software Fault Tolerance, its relevance, and its relation to other approaches in constructing reliable software. Steve Feiner’s insightful reading and comments improved the paper.

This work was supported in part by equipment grants from AT&T and the Hewlett-Packard Corporation, and in part by NSF grant CDR-84-21402.

8. References

[Anderson1981a] T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice*, Prentice-Hall International (1981).

[Anderson1982a] T. Anderson and P.A. Lee, ''Fault Tolerance Terminology Proposals,'' in *Proceedings, 12th International Symposium on Fault-Tolerant Computing*, Los Angeles, CA. (June 1982), pp. 29-33.

[Anderson1986a] T. Anderson, ''A structured decision mechanism for Diverse Software,'' in *Proceedings, IEEE 1986 Symposium on Reliability in Distributed Software and Database Systems.* (1986), pp. 125-129.

[Avizienis1977a] A. Avizienis and L. Chen, ''On the implementation of N-version programming for software fault tolerance during execution,'' in *Proceedings, COMPSAC 77, 1st IEEE-CS International Computer Software and Applications Conference*, Chicago, IL (November 8-11 1977), pp. 149-155.

[Avizienis1978a] A. Avizienis, ''Fault tolerance: The survival attribute of digital systems,'' *Proceedings of the IEEE* **66**, pp. 1109-1125 (October 1978).

[Avizienis1984a] A. Avizienis and John P.J. Kelly, ''Fault Tolerance by Design Diversity: Concepts and Experiments,'' *IEEE Computer*, pp. 67-80 (August 1984).

[Avizienis1985a] A. Avizienis, ''The N-Version Approach to Fault-Tolerant Software,'' *IEEE Transactions on Software Engineering*, pp. 1491-1501 (December 1985).

[Avizienis1985b] A. Avizienis, P. Gunningberg, J. P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso, and U. Voges, ''The UCLA DEDIX system: a Distributed Testbed for Multiple-Version Software,'' in *Digest of FTCS-15, the 15th International Symposium on Fault-Tolerant Computing*, Ann Arbor, Michigan (June 1985), pp. 126-134.

[Brooks1975a] F. P. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley, Reading, Mass. (1975).

[Chen1978a] L. Chen and A. Avizienis, ''N-version programming: A fault tolerance approach to reliability of software operation,'' in *Digest, 8th Annual International Conference on Fault-Tolerant Computing*, Toulouse, France (June 21-23 1978), pp. 3-9.

[Dahl1972a] O.-J. Dahl, C.A.R. Hoare, and E.W. Dijkstra, *Structured Programming*, Academic Press, New York (1972).

[Dijkstra1972a] E.W. Dijkstra, ''Notes on Structured Programming,'' in *Structured Programming*, Academic Press, New York (1972).

[Dijkstra1976a] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J. (1976).

[Eckhardt1985a] Dave E. Eckhardt, Jr. and Larry D. Lee, ''A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors,'' *IEEE Transactions on Software Engineering* **SE-11**(12), pp. 1511-1517 (December 1985).

[Feynman1963a] Richard P. Feynman, Robert B. Leighton, and Matthew Sands, *The Feynman Lectures on Physics*, Addison-Wesley, Reading, MA (1963).

[Floyd1967a] R.W. Floyd, ''Assigning meanings to programs,'' in *Proceedings, American Math. Society Symposium in Applied Mathematics* (1967), pp. 19-31.

[Gerhart1976a] Susan L. Gerhart and Lawrence Yelowitz, ''Observations of Fallibility in Applications of Modern Programming Methodologies,'' *IEEE Transactions on Software Engineering* (September 1976).

[Goel1979a] A.L. Goel and K. Okumoto, ''A time dependent error detection rate model for software reliability and other performance measures,'' *IEEE Transactions on Reliability*, pp. 206-211 (August 1979).

[Hansen1983a] R.C. Hansen, R.W. Peterson, and N.O. Whittington, ''Fault Detection and Recovery,'' *Bell System Technical Journal* **62**(1), pp. 349-366 (January 1983).

[Hecht1979a] H. Hecht, ''Fault-Tolerant Software,'' *IEEE Transactions on Reliability*, pp. 227-232 (August 1979).

[Hoare1969a] C.A.R. Hoare, ''An axiomatic basis for computer programming,'' *Communications of the ACM* **12**, pp. 576-580, 583 (October 1969).

[Horning1974a] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell, ''A program structure for error detection and recovery.,'' in *Proceedings, Conference on Operating Systems: Theoretical and Practical Aspects* (April 1974), pp. 177-193.

[Knight1985a] John C. Knight, Nancy G. Leveson, and Lois D. St. Jean, ''A Large Scale Experiment in N-Version Programming,'' in *Proceedings of the 15th Annual International Symposium on Fault-Tolerant Computing (FTCS-15)*, IEEE (1985), pp. 135-139.

[Knight1986a] J.C. Knight and N.G. Leveson, ''An Empirical study of failure probabilities in multi-version software.,'' in

Proceedings of the 16th Annual International Symposium on Fault-Tolerant Computing (FTCS-16), Vienna, Austria (July 1986), pp. 165-170.

[Knight1986b] J.C. Knight and N.G. Leveson, ''An Experimental Evaluation of the Assumption of Independence in Multiversion Programming,'' *IEEE Transactions on Software Engineering* **SE-12**(1), pp. 96-109 (January 1986).

[Koenig1986a] Andrew Koenig, ''C Traps and Pitfalls,'' Computing Science Technical Report No. 123 (July 1, 1986). AT&T Bell Laboratories

[Laprie1985a] Jean-Claude Laprie, ''Dependable Computing and Fault Tolerance: Concepts and Terminology,'' in *Fifteenth Annual International Symposium on Fault-Tolerant Computing*, Ann Arbor, MI (June 1985), pp. 2-11.

[Lee1980a] P.A. Lee, N. Ghani, and K. Heron, ''A Recovery Cache for the PDP-11,'' *IEEE Transactions on Computers* **C-29**(6), pp. 546-549 (June 1980).

[Lee1982a] P.A. Lee and D.E. Morgan, ''Fundamental Concepts of Fault Tolerant Computing, Progress Report,'' in *Proceedings, 12th International Symposium on Fault-Tolerant Computing*, Los Angeles, CA. (June 1982), pp. 34-38.

[Lerner1988a] Mark D. Lerner, ''Fault Tolerance on Massively Parallel Processors,'' Technical Report #CUCS-370-88, Columbia University Computer Science Department (1988).

[Leveson1986a] Nancy G. Leveson, ''Software Safety: What, Why, and How,'' *ACM Computing Surveys* **18**(2), pp. 125-163 (June, 1986).

[Littlewood1979a] B. Littlewood, ''How to Measure Software Reliability and How Not To,'' *IEEE Transactions on Reliability*, pp. 103-110 (June 1979).

[Littlewood1979b] B. Littlewood, ''Software Reliability Model for Modular Program Structure,'' *IEEE Transactions on Reliability* (August 1979).

[Melliar-Smith1979a] P.M. Melliar-Smith, ''System Specification,'' in *Computing Systems Reliability*, ed. B. Randell, Cambridge University Press (1979), pp. 19-65.

[Morgan1982a] David E. Morgan, ''Report of Subcommittee on Models, Fundamental Concepts, and Terminology,'' in *Proceedings, 12th International Symposium on Fault-Tolerant Computing*, Los Angeles, CA. (June 1982), pp. 3-5.

[Musa1975a] John D. Musa, ''A theory of software reliability and its application,'' *IEEE Transactions on Software Engineering* **SE-1**, pp. 312-327 (September 1975).

[Musa1979a] John D. Musa, ''Validity of Execution-Time Theory of Software Reliability,'' *IEEE Transactions on Reliability* (August 1979).

[Musa1987a] John D. Musa, Anthony Iannino, and Kazuhira Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill (1987).

[Neumann1956a] John von Neumann, ''Probabilistic logics and the synthesis of reliable organisms from unreliable components,'' in *Automata Studies*, ed. J. McCarthy, Princeton University Press (1956), pp. 43-98.

[Parnas(null)a] D. L. Parnas, ''On the Criteria to be used in Decomposing Systems into Modules,'' *Communications of the ACM* **15**(12).

[Prather1983a] R.E. Prather, ''Theory of Program Testing - An Overview,'' *The Bell System Technical Journal* **62**(10, part 2), pp. 3073-3105 (December 1983).

[Ramamoorthy1982a] C. V. Ramamoorthy and Farokh B. Bastani, ''Software Reliability - Status and Perspectives,'' *IEEE Transactions on Software Engineering* **SE-8**(4), pp. 354-371 (July 1982).

[Randell1975a] B. Randell, ''System structure for software fault tolerance,'' *IEEE Transactions on Software Engineering* **SE-1**, pp. 220-232 (June 1975).

[Robbins1975a] Herbert Robbins and John Van Ryzin, *Introduction to Statistics*, SRA (1975).

[Scott1983a] R. Keith Scott, James W. Gault, and David F. McAllister, ''Modeling Fault-Tolerant Software Reliability,'' in *Proceedings, IEEE 1983 Symposium on Reliability in Distributed Software and Database Systems* (1983), pp. 15-27.

[Scott1983b] Roderick Keith Scott, ''Data Domain Modeling of Fault-Tolerant Software Reliability,'' Ph.D. Thesis, North Carolina State University at Raleigh (1983).

[Scott1984a] R. Keith Scott, James W. Gault, David F. McAllister, and Jeffrey Wiggs, ''Experimental Validation of Six Fault-Tolerant Software Reliability Models,'' in *Proceedings of the 14th Annual International Symposium on Fault-Tolerant Computing* (1984).

[Shrivastava1985a] Santosh K. Shrivastava, *Reliable Computing Systems*, Springer-Verlag (1985).

[Siewiorek1982a] Daniel P. Siewiorek and Robert S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press (1982).

[Smith1987a] Jonathan M. Smith and Gerald Q. Maguire,Jr., ‘‘RB: Programmer Specification of Redundancy,’’ Technical Report CUCS-269-87, Columbia University Computer Science Department (1987).

[Strom1983a] R. E. Strom and S. Yemini, ‘‘NIL: An Integrated Language and System for Distributed Programming,’’ *ACM SIGPLAN Notices*, pp. 73-82 (June 1983).

[Toy1983a] W.N. Toy and L.E. Gallaher, ‘‘Overview and Architecture of the 3B20D Processor,’’ *Bell System Technical Journal* **62**(1), pp. 181-190 (January 1983).