# A Comparative Analysis of Hardware and Software Fault Tolerance: Impact on Software Reliability Engineering

H. Ammar, B. Cukic, C. Fuhrman, and A. Mili
Institute for Software Research
Fairmont, WV 26554 USA
http://www.isr.wvu.edu

January 15, 1999

# Contents


**1 Introduction to Fault Tolerance** **3**
1.1 Coping with Faults . . . 3
1.2 Fault, Error and Failure . . . 3
1.3 Fault Tolerance . . . 5
1.4 Organization . . . 6

**2 Modeling Fault Tolerance** **6**
2.1 Modeling Failure . . . 6
2.2 Modeling Errors . . . 8
2.2.1 Locus of Control . . . 8
2.2.2 Past and Future . . . 10
2.2.3 Levels of Correctness . . . 11
2.2.4 Levels of Recoverability . . . 12
2.3 Modeling Faults . . . 15
2.3.1 Logical Modeling . . . 15
2.3.2 Statistical Modeling . . . 16
2.4 Illustrative Examples . . . 19
2.4.1 A Hardware Example: The Pentium ™Processor FDIV fault . . . 19
2.4.2 A Software Example: The Ariane 5 Exception Handling Fault . . . 20

**3 Deploying Fault Tolerance** **21**
3.1 Error Detection . . . 21
3.2 Damage Assessment . . . 22
3.3 Error Recovery . . . 23
3.4 Fault Removal . . . 25

**4 Design of Fault Tolerant Systems** **26**
4.1 Redundancy as the Basis of Fault Tolerance . . . 26
4.2 Operating Cost vs. Temporal Redundancy . . . 27
4.3 Development Cost vs Spatial Redundancy . . . 28
4.4 Design Complexity vs Informational Redundancy . . . 29

**5 Impact of Fault Tolerance on Reliability** **30**
5.1 Hardware Reliability Modeling . . . 30
5.1.1 Reliability Modeling Techniques . . . 31
5.1.2 Combinatorial Models . . . 31
5.1.3 Markov Models . . . 31
5.2 Software Reliability Modeling . . . 31
5.2.1 Input-Domain Modeling . . . 32
5.2.2 Time-Domain Modeling . . . 33
5.3 Reliability Measurement . . . 34

**6 Conclusion** **35**
6.1 Summary . . . 35
6.2 Lessons Learned . . . 35

# 1 Introduction to Fault Tolerance

## 1.1 Coping with Faults

Generally speaking, a fault is a feature of a system that precludes it from operating according to its specification. Methods for coping with the existence and manifestation of faults can be classified into three families:

- *Fault avoidance*, which includes methods that are geared towards the design of systems that are provably free of faults. Typical software systems are generally so complex that it is virtually impossible to achieve fault free software products. On the other hand, hardware faults typically stem not from poor design but from physical wear and operating conditions. Hence fault avoidance is unrealistic, for software as well as for hardware.

- *Fault removal*, which recognizes that products may have faults, and aims to remove them after the product development is completed. For the same reasons as above, the goal of fault removal is not realistic in practice, in the sense that no assurance can be given for the total absence of faults after fault removal.

- *Fault tolerance*, which recognizes that neither fault avoidance not fault removal can ensure the absence of faults, and aims to make provisions for operating under the hypothesis that the system has faults. Provisions to be considered include the detection of errors caused by faults and the recovery from these errors as well as the identification and removal of the faults.

While it makes for colorful discussions to debate the relative merits of each method, it is wise to recognize that each family of methods is best adapted in some set of circumstances, and that by virtue of the law of diminishing returns, it is best to use all three families of methods in concert. In this paper, we focus on methods of fault tolerance, and investigate the differences between hardware fault tolerance and software fault tolerance.

## 1.2 Fault, Error and Failure

In this section we briefly present working definitions of fault, error and failure; these concepts will be discussed in greater detail in the sequel. Whenever it is possible, we will provide general definitions, that apply equally well for hardware and for software.

**Definition 1** *A* fault *is a feature of a system that precludes it from operating according to its specification.*

While this definition applies equally well to hardware systems and software systems, we illustrate it here with an example from software. We consider the variable $s$ of type **natural** and we consider the following program, which is expected to compute the remainder by 3 of the square of the input value.

```
program modthreeofsquare;
   begin
   read(s);
   s:= 2*s;   {<------ fault}
   s:= s mod 3;
   write(s)
   end.
```

While the program is *expected* to compute the remainder by 3 of the square of the input value, what it *actually* computes is the remainder by 3 of twice the input value.

**Definition 2** *We say that we have observed an* error *at some step $L$ of a computation if and only if the value of the system state at step $L$ differs from the expected value at that step.*

If we consider the program presented above and let $L$ be the label shown on the program text, then we find that for the initial value $s_0 = 3$ the state at label $L$ is $s = 6$ whereas the expected value at that label is $s = 9$; this is an error.

```
program modthreeofsquare;
   begin                {expected state       actual state}
   read(s);             {s =  3               s = 3       }
   s:= 2*s;             {     9                   6       }
   L:                   {     9         =/=       6   <------ error}
   s:= s mod 3;
   write(s)
   end.
```

Note that if the initial state is $s_0 = 2$, then the fault does not cause an error, since $2 \times 2 = 2^2 = 4$. We say that an initial state *sensitizes* a fault if and only if the state of the system at label $L$ in the presence of the fault differs from the state in the absence of the fault.

**Definition 3** *We say that we have observed a* failure *of a system if and only if the actual output of the system for some input differs from the expected output.*

Failure is, in fact, nothing but a fault at label `end`. If we consider again the example of the program given above, then neither input 2 nor input 3 causes a failure, since 2 does not even cause an error, and 3 produces 6 instead of 9 at label $L$, both of which return 0 after taking the remainder by 3. For input $s_0 = 4$, however, the program returns $(2 \times 4)$ `mod` $3 = 2$ whereas the expected value is $(4^2)$ `mod` $3 = 1$. Whenever the error causes a failure (as was the case for $s_0 = 4$), we say that the error has been *propagated* to the output. When the error does not cause a failure (as was the case for $s_0 = 3$), we say that the error has been *masked*.

```
program modthreeofsquare;
   begin                {expected state       actual state}
   read(s);             {s =   4              s = 4       }
   s:= 2*s;             {     16                  8       }
   L:                   {     16                  8       }
   s:= s mod 3;         {      1                  2       }
   write(s)             {      1         =/=      2   <------ failure}
   end.
```

To highlight again that these definitions of fault, error and failure apply equally well to software and hardware, we present below an example borrowed from hardware fault tolerance, as a complement to the sample program discussed above. We consider the pattern of *triple modular redundancy*, whereby three separate circuits compute some function $F$, and a voter checks that they produce the same output for any input; if two out of three produce the same output, the voter selects that as the correct output. If all three produce different outputs, the voter raises an exception, whose outcome is to consider the output void. We consider the circuit of figure 1. In the example shown in figure 1, the fault is the fact that the second and third modules compute functions $F'$ and $F''$ rather than the expected function $F$. An error arises at cut (label) $L$ if and only if, for a given input $X$, we fail to have the following condition:

$$F(X) = F'(X) = F''(X).$$

If only one faulty module deviates from $F$ (e.g. $F(X) = F'(X)$ and $F(X) \neq F''(X)$) then the voter will select the majority value, which is $Y = F(X)$, yielding a correct output —hence no failure occurs in this case. If, for input $X$, $F'$ and $F''$ yield the same output, which is faulty (i.e. $F'(X) = F''(X)$ and $F'(X) \neq F(X)$) then the voter will select the output $Y = F'(X)$, thereby leading to a failure. Also, if all three values are distinct, then the voter declares a failure and its data output is undefined.

Note that while we have carried out this discussion under the hypothesis that all modules are expected to produce the same output, a more realistic (and more general) hypothesis is to consider that they must be equivalent (in some sense) rather than equal. Brilliant et al [6] and Lorczak et al [19] have investigated forms of equivalence relation that are appropriate for this purpose, and their application to modular redundancy.

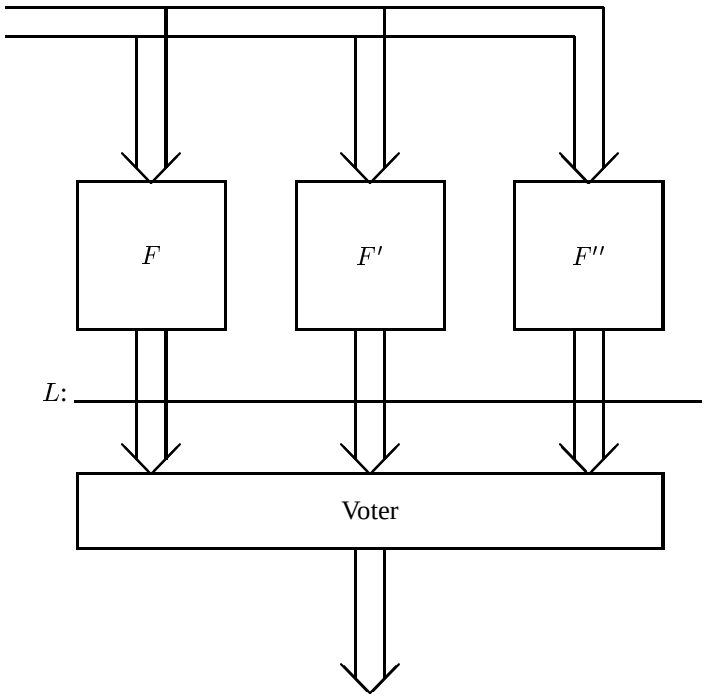


Figure 1: An Example of Fault, Error and Failure in Hardware

## 1.3 Fault Tolerance

The definitions given above allow us to define *fault tolerance*.

**Definition 4** *A system is said to be* fault tolerant *if and only if it has provisions for avoiding failure after faults have caused errors.*

In fault tolerance, we resign ourselves to the presence of faults in the system, and we take measures to ensure that faults do not cause failure. Such measures include the following steps:

- *Detecting Errors*. Error detection is primarily dependent on redundancy, including natural redundancy (e.g. error detecting codes in hardware, executable assertions in software) and artificial redundancy (e.g. modular redundancy in hardware and N-version programming in software).
- *Assessing Damage*. This step involves assessing the extent of the damage caused by the fault on the state of the computation, and making a decision on what recovery action must be taken.
- *Error Recovery*. This step involves correcting the current state of the computation according to the findings of the damage assessment step, and resuming the computation.
- *Fault Removal*. While the three preceding steps deal with the current computation and are performed on-line for the sake of salvaging the current computation, this step deals with the system itself, and can be performed off-line. It consists of identifying the fault that caused the error and removing it.

The three first steps deal with the error, whereas the last step deals with the fault. Design issues related to fault tolerance include the selection of labels where error detection is performed, the derivation of detection formulas, and the deployment of damage assessment and error recovery procedures.

### 1.4 Organization

In the sequel, we consider in turn all the definitions we have presented above about fault tolerance, and analyze them in turn, especially with respect to how they apply differently to hardware and software. In section 2 we consider in turn the three fundamental concepts of fault tolerance (failure, error, fault) and analyze them comparatively for hardware and software. In section 3, we consider in turn the four steps of fault tolerance (error detection, damage assessment, error recovery, and fault removal) and analyze them comparatively for hardware and software. Then we consider the impact of the fault tolerance requirement on system design (the process) and system quality (the product): specifically, in section 4 we analyze the impact of fault tolerance requirements on system design and in section 5 we analyze the impact of fault tolerance requirements on system reliability. In the conclusion (section 6) we summarize our findings and draw some general lessons on hardware and software fault tolerance.

## 2 Modeling Fault Tolerance

The definition of fault tolerance (definition 4) highlights three key concepts, failure, error and fault, which we propose to discuss in this section.

### 2.1 Modeling Failure

According to definition 3, we say that we have observed a *failure* of a system if and only if the actual output of the system for some input differs from the expected output. This definition raises two issues, which we must address in some detail: first, it appears that, for the purposes of fault modeling, system behavior can be modeled by input/outputs; second, there appears to be some well defined expected behavior of the system in question. We take a closer look at these issues.

We resolve to capture the functional behavior of a system (hardware or software) by a function from inputs to outputs, i.e. a set of input/output pairs. While it is easy to see how an adder (hardware) or a sort routine (software) can be represented by input/output pairs, it may be difficult to see how a counter (hardware) or an operating system (software) can. In fact, we have shown in [34] how state-bearing objects (such as counters, operating systems, command and control systems) can be represented by pairs of the form $(h, y)$, where $h$ is a sequence of inputs and $y$ is an output: by redefining the input space as a set of input sequences, we can model system function as a set of input/output pairs. As for the issue of *expected behavior* of a system, we submit that there are two ways to capture expected behavior, which result in two distinct definitions of failure.

- *The function that the system is known to compute*. We consider an adder of two signed integers, where each integer is represented over 64 bits, and we assume that we are using this adder to add natural numbers. If, for inputs +20 and +55 we obtain the output 75 but we notice that the sign bit is negative, then we conclude that there is a failure, because we know that the adder should have returned +75. Also, consider the case of a routine that computes the greatest common divisor of two positive integers $x$ and $y$ by Euclid's method and returns the output in $x$. If for inputs 20 and 55 we find that the final value of $x$ is 5 but the final value of $y$ is, e.g. 11, then we conclude that there is a failure because Euclid's methods should have returned the same value in $x$ and $y$.

- *The specification that the system is built to satisfy*. Under this view, we have no cognizance of the function that the system is supposed to compute, but we focus on the specification that the system is supposed to satisfy. We consider that there is failure only if the output fails to satisfy the requirements dictated by the specification. In both examples cited above (the adder circuit and the GCD routine) the sample executions raise no failure with respect to their specifications (although they raise failure with respect to their expected functions).

This dichotomy leads to two different definitions of failure, which we present below. We consider a system $\sigma$ which is known to compute a function $\phi$ from some space $S$ to some space $Z$ (i.e. $\phi \subseteq S \times Z$).

**Definition 5** *We say that we have observed a* failure of system $\sigma$ with respect to its function $\phi$ *if and only if for some initial state $s$ the corresponding output state $s'$ does not satisfy the condition $(s, s') \in \phi$.*

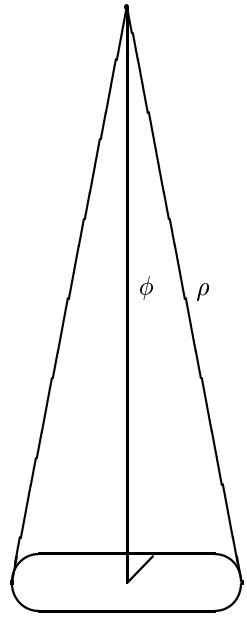


Figure 2: Degrees of Failure

For the sake of illustration, we consider the examples of adder and GCD discussed above and define their function. If we let $x$, $y$ and $z$ be variables of type `signed64bitinteger`, then the function $A$ defined by the adder can be written as:

$$A = \{(\langle x, y\rangle, z) | z = x + y\}.$$

On the other hand, if $x$ and $y$ are variables of type **integer**, then the function of the GCD routine can be written as:

$$G = \{(\langle x, y\rangle, \langle x', y'\rangle) | x' = gcd(x, y) \wedge y' = gcd(x, y)\}.$$

**Definition 6** *We say that we have observed a* failure of system $\sigma$ with respect to specification $\rho$ *if and only if for some initial state $s$ the output $s'$ returned by $\sigma$ does not satisfy the condition $(s, s') \in \rho$.*

For the sake of illustration, we consider the following specifications for the adder circuit and the GCD routine, which we denote respectively by $\alpha$ and $\gamma$.

$$\alpha = \{(\langle x, y\rangle, z) |\ x \geq 0 \wedge y \geq 0 \wedge |z| = x + y\}.$$

$$\gamma = \{((x, y), (x', y')) | x' = gcd(x, y)\}.$$

Note that the pair $(\langle +20, +55\rangle, -75)$ satisfies relation $\alpha$, while it is not in $A$; likewise, the pair $(\langle 20, 55\rangle, \langle 5, 7\rangle)$ satisfies relation $\gamma$, while it is not in $G$. The contrast between failure with respect to the system function and failure with respect to the system specification is illustrated in figure 2.

To be perfectly exhaustive, we need, actually, to add another level of failure: in addition to failure with respect to the system's function, and failure with respect to the system's specification, we may want to consider failure of the system's requirements —i.e. take into account the case where the specification itself is at fault and fails to capture its intended purpose. This is not only possible, it is actually quite common [18]. Nevertheless, we choose to exclude specification failure from our discussions, because it is difficult to model them, and we wish to keep the discussions of this paper focused.

## 2.2 Modeling Errors

Definition 4 provides that fault tolerance is the ability of a system to avoid failure after faults have caused errors. Implicit in this definition is the premise that errors are detected before failure occurs —the sooner the better, in fact, since we want to intervene before the system state is damaged too extensively. Hence fault tolerance rests on the ability to check for anomalies at predefined points along the computation process; these points are the subject of our subsequent discussion.

### 2.2.1 Locus of Control

For the purposes of this discussion, it is best to view a computation (whether it is done in hardware or in software) as a sequence of function invocations where each function receives input data from the preceding function in the sequence and feeds its output to the next function. Consider the following program that multiplies natural variables $x$ and $y$ into variable $z$.

```
begin
read(x,y);
z:= 0;
while L: y<>0 do
   begin
   if y mod 2 = 0 then
      begin
      y:= y div 2;
      x:= 2*x
      end
   else
      begin
      y:= y-1;
      z:= z+x
      end
   end;
write(z)
end.
```

We place label $L$ before the test of the loop condition; this label corresponds to the position shown in the flowchart of figure 3. Because label $L$ can potentially be visited more than once, we cannot define the locus of control by $L$ alone but rather by a pair of the form $(v, L)$, where $v$ is the ordinal of the visit to $L$; we refer to the pair $(v, L)$ as a *milestone*.

In hardware, the identification of the locus of control is more difficult, because the distinction between data and control is more subtle. For simple combinatorial devices, any cut across the circuit can be considered an intermediate locus of control. For (the more common) sequential devices, we define the locus of control by considering the sequence of transformations (called operation codes, micro-instructions, or instructions, depending on their level of abstraction). For the sake of illustration, we consider the hardware device that multiplies two unsigned integers in plain binary representation by shift and add (the hardware equivalent of the program discussed above, in fact). The data and control paths for this device (due to [36], page 213), are shown in figure 4; control paths are represented by single arrows, labeled with the operation code (+ for addition, $\Rightarrow$ for shift), and data paths are represented by double arrows. The locus of control can be defined as a point in the sequence of Shift/Add commands that the control logic sends to the adder and the registers. Because the same control point can be visited a number of times in any given execution (e.g. successive Shift/Add cycles), we again define the locus of control by a pair of the form $(v, L)$ where $L$ is a label in the control code and $v$ is the ordinal of the visit to $L$.

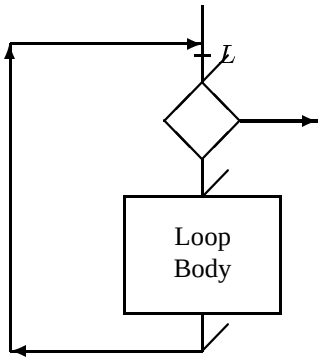


Figure 3: Labeling a While Loop

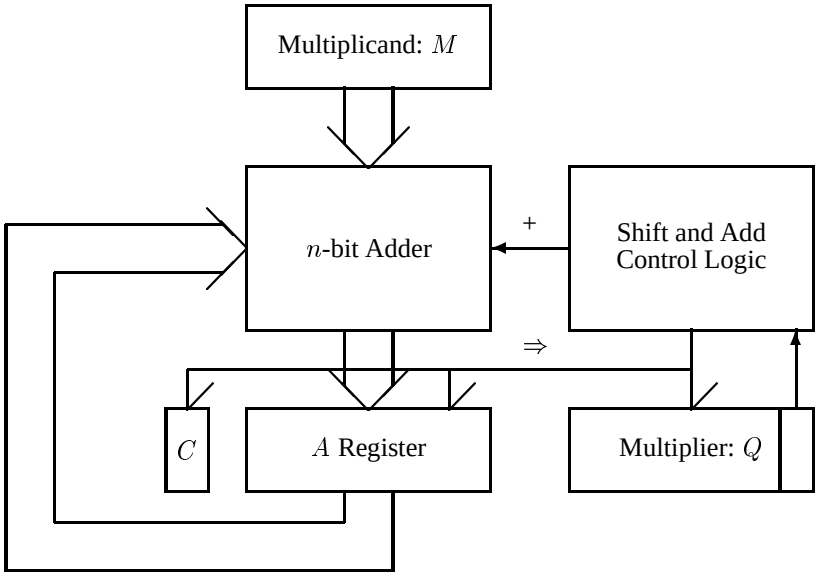


Figure 4: Shift and Add Circuit

### 2.2.2 Past and Future

We use the concept of *milestone* to define functions *past* and *future*. For the sake of readability, we choose to present these definitions in the context of software, even though they apply equally well to hardware. The reason why software offers a better background for these definitions is that it affords us a clearer distinction between control and data: control is represented by executable statements and data is represented by variable declarations. By contrast, hardware blurs this distinction somewhat: consider that in the multiplication circuitry of figure 4, register $Q$ can be considered as data (multiplier) but also as control (the bits of $Q$ are used to control the shift and add device). The distinction between control and data is central to our discussion, since faults are features of the control whereas errors are features of the data.

**Definition 7** *The* past path *of a program $P$ at milestone $m$ is the sequence of statements that are executed prior to reaching $m$. The* past function *of program $P$ at milestone $m$ is the function denoted by $\Pi_P^m$ and defined as the function computed by the past path of program $P$ at milestone $m$.*

We consider the multiplication program presented earlier, defined on natural variables $x$, $y$ and $z$.

```
begin
z:= 0;
while L: y<>0 do
   begin
   if y mod 2 = 0 then
      begin
      y:= y div 2;
      x:= 2*x
      end
   else
      begin
      y:= y-1;
      z:= z+x
      end
   end;
end.
```

At milestone $m = (3, L)$, the past path is defined as follows:

```
z:=0;
(1,L): ((y<>0)!); B;
(2,L): ((y<>0)!); B;
(3,L):
```

where `B` refers to the loop body, and `((y<>0)!)` refers to the event of testing condition `(y<>0)` and finding it to hold. The past function can be derived by taking the product of all the terms in the sequence of the past path, using the following formula for the function of `((y<>0)!)`:

$$\{(\langle x, y, z\rangle, \langle x', y', z'\rangle) | y \neq 0 \wedge x' = x \wedge y' = y \wedge z' = z\}.$$

By analyzing the statements that appear in the past path and composing their function, we find the following function as the past function of this program at milestone $m = (3, L)$:

$$\begin{aligned}
&\Pi_P^m \\
&= \{(\langle x, y, z\rangle, \langle x', y', z'\rangle) | y > 2 \wedge y \texttt{ mod } 4 = 0 \wedge y' = y \texttt{ div } 4 \wedge x' = 4 \times x \wedge z' = z\} \\
&\cup \{(\langle x, y, z\rangle, \langle x', y', z'\rangle) | y > 2 \wedge y \texttt{ mod } 2 = 0 \wedge (y \texttt{ div } 2) \texttt{ mod } 2 = 1 \wedge \\
&\qquad\qquad y' = (y \texttt{ div } 2) - 1 \wedge x' = 2 \times x \wedge z' = z + 2 \times x\} \\
&\cup \{(\langle x, y, z\rangle, \langle x', y', z'\rangle) | y > 2 \wedge y \texttt{ mod } 2 = 1 \wedge x' = 2 \times x \wedge y' = (y - 1) \texttt{ div } 2 \wedge z' = z + x\}.
\end{aligned}$$

**Definition 8** *The* future path *of a program $P$ at milestone $m$ is the sequence of statements that are executed downstream of $m$. The* future function *of program $P$ at milestone $m$ is the function denoted by $\Phi_P^m$ and defined as the function computed by the future path of program $P$ at milestone $m$.*

We consider the multiplication program discussed above and find that its future path can be written as the following sequence of statements, where `((y<>0)!)` denotes testing condition `(y<>0)` and finding it to hold, and `((y<>)-)` denotes testing the condition and finding it not to hold.

```
(3,L): (y<>0)!; B;
...  ...  ...
(v,L): (y<>0)!; B;
...  ...  ...
(n,L): (y<>0)-;
```

Of course, the test `(y<>0)` may very well fail (evaluate to false) the very first time it is checked (at milestone $(3, L)$). This sequence of statements consists in effect in iteratively testing the condition `(y<>0)` and executing `B` until the test returns false: this is nothing but the while loop of program $p$. Hence the future path can be written as:

```
(3,L):  while y<>0 do B;
```

By analyzing the statements that appear in the past path and composing their function, we find that the future function at milestone $m = (3, L)$ is:

$$\Phi_P^m = \\ \{(\langle x, y, z\rangle, \langle x', y', z'\rangle) | x' = g(x, y) \wedge y' = 0 \wedge z' = z + x \times y\},$$

for some function $g$.

Functions past and future can be used to define three concepts: what it means to *sensitize* a fault; what it means to *propagate* an error; and what it means to *mask* an error. We discuss these in the following definition. We consider a program $P$, a milestone $m$ on $P$, and an initial state $s_0$ of $P$. We let $s_m$ be the expected state at milestone $m$ and $s_F$ the expected state at the end of the execution of $P$; we assume that the past path of program $P$ at milestone $m$ is potentially faulty.

**Definition 9** *We say that a fault in the past path of $P$ at $m$ is* sensitized *for input $s_0$ if and only if $\Pi_P^m(s_0) \neq s_m$. Given that the state at milestone $m$, $s = \Pi_P^m(s_0)$, is erroneous, we say that the error is* propagated *if and only if $\Phi_P^m(s) \neq s_F$; we say that the error is* masked *if and only if $\Phi_P^m(s) = s_F$.*

The example discussed in section 1 presents instances of sensitization, propagation, and masking.

### 2.2.3 Levels of Correctness

As a preliminary to our discussion on levels of correctness, we briefly present some relational notations: the product of two relations $R$ and $R'$ is denoted by $R \circ R'$, or by $RR'$ if this raises no ambiguity; the inverse of relation $R$ is denoted by $\widehat{R}$; and the set of images of $s$ by relation $R$ is denoted by $R(s)$ (by generalization of the functional notation).

In the same way as we have defined levels of failure, we define here levels of error —or conversely, levels of correctness, at some milestone $m$. We consider a program $P$, a milestone $m$ on $P$, and an initial state $s_0$ of $P$. We denote by $\Delta_P^m$ the *expected* past function of $P$ at $m$: because $P$ may be faulty (in particular, it may contain faults in the past path of program $P$ at milestone $m$), the *actual* past function $\Pi_P^m$ may differ from the expected past function $\Delta_P^m$.

**Definition 10** *A state $s$ is said to be* correct *at milestone $m$ for initial state $s_0$ if and only if $s = \Delta_P^m(s_0)$.*

For a given initial state and milestone, there exists a single correct state. The next definition depends on the expected program function, which we denote by $\phi$.

**Definition 11** *A state $s$ is said to be* F-maskable *at milestone $m$ for initial state $s_0$ if and only if $\Phi_P^m(s) = \phi(s_0)$.*

The set of *F-maskable* states at milestone $m$ is the image set of $s_0$ by the relation $\phi \circ \widehat{\Phi_P^m}$. Because function $\Phi_P^m$ is typically non injective (two or more arguments may have the same image by $\Phi_P^m$), relation $\widehat{\Phi_P^m}$ is typically non deterministic (the same argument may have two or more images by $\widehat{\Phi_P^m}$); hence yielding a potentially large set of F-maskable states. See figure 5. The next definition depends on the program specification, which we denote by $\rho$.

**Definition 12** *A state $s$ is said to be* S-maskable *at milestone $m$ for initial state $s_0$ if and only if $\Phi_P^m(s) \in \rho(s_0)$.*

The set of *S-maskable* states at milestone $m$ is the image set of $s_0$ by the relation $\rho \circ \widehat{\Phi_P^m}$. Because function $\Phi_P^m$ is typically non injective, relation $\widehat{\Phi_P^m}$ is typically non deterministic; this, combined with the nondeterminacy of relation $\rho$, yields a potentially large set of S-maskable states. See figure 5.

### 2.2.4 Levels of Recoverability

A state of a program may well fail to meet any of the conditions discussed in the definitions above, and yet still contain all the information that is required for a successful (failure-free) completion of the computation. As an example, consider a selection sort algorithm, in which the index has jumped too far to the right. For example, the initial array is $a_0 = [6, 5, 7, 4, 9, 3, 2]$ and at the second iteration through the selection sort, the array contains the sequence $[2, 3, 7, 4, 9, 5, 6]$ and the index is equal to 5 (rather than 3). This state is not correct, because the index does not have the correct value. It is not F-maskable either because subsequent execution of the sort routine on this state will not produce the expected result; also, if we let $\rho$ be a specification that requires the output to be sorted, then this state is not S-correct with respect to $\rho$ either. Yet, this state contains all the information that is required to produce a failure free final state (by contrast with a state where, e.g. some cells of the array are overwritten). We then say that this state is *recoverable*; in the sequel we briefly discuss three definitions of recoverability. All these definitions hinge on what it means for a state to *contain all the information required to satisfy some condition*; we understand that to mean that there exists a function which, when applied to the current state, produces a state that satisfies the condition. Whence the following definitions.

**Definition 13** *A state $s$ is said to be* recoverable *at milestone $m$ if and only there exists a function $r_m$ such that $r_m(s)$ is correct at milestone $m$.*

Of course, function $r_m$ must be independent of $s$, otherwise the definition is meaningless; the function does depend on $m$, which is why we index it therewith. The set of recoverable states at milestone $m$ is the set of images of $s_0$ by the relation $\Delta_P^m \circ \widehat{r_m}$.

**Definition 14** *A state $s$ is said to be* F-recoverable *at milestone $m$ if and only there exists a function $r_m$ such that $r_m(s)$ is F-maskable at milestone $m$.*

The set of F-maskable states at milestone $m$ is the set of images of $s_0$ by the relation $(\phi \circ \widehat{\Phi_P^m} \circ \widehat{r_m})$. This set grows larger as $\Phi_P^m$ and $r_m$ grows less injective (more arguments have the same image).

**Definition 15** *A state $s$ is said to be* S-recoverable *at milestone $m$ if and only there exists a function $r_m$ such that $r_m(s)$ is S-maskable at milestone $m$.*

The set of S-maskable states at milestone $m$ is the set of images of $s_0$ by the relation $(\rho \circ \widehat{\Phi_P^m} \circ \widehat{r_m})$. This set grows larger as $\rho$ grows less deterministic (more images for a given argument) and as $\Phi_P^m$ and $r_m$ grow less injective (more arguments have the same image). The three instances of function $r_m$ mentioned in the definitions above are not necessarily identical. Figures 6 and 7 show how these levels of recoverability relate to the levels of correctness and maskability discussed above.

Interestingly, figure 6 can be seen as giving a meaning to *damage assessment* (re: section 1). To assess the damage in a state $s$ consists in effect in determining where $s$ lies with respect to the boundaries drawn on this figure; this matter will be discussed further in section 3.2.

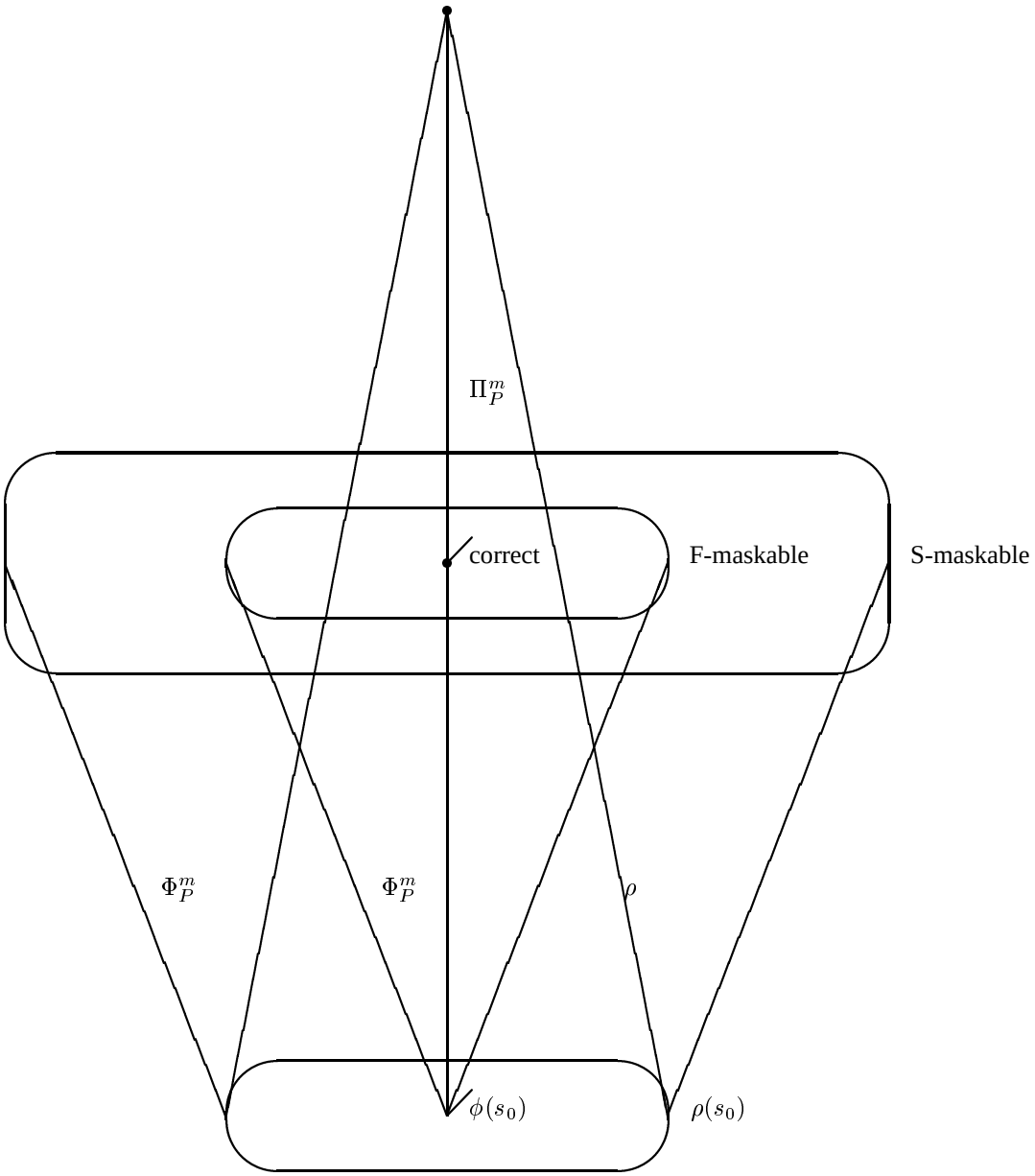


Figure 5: Degrees of Error: Correctness and Maskability

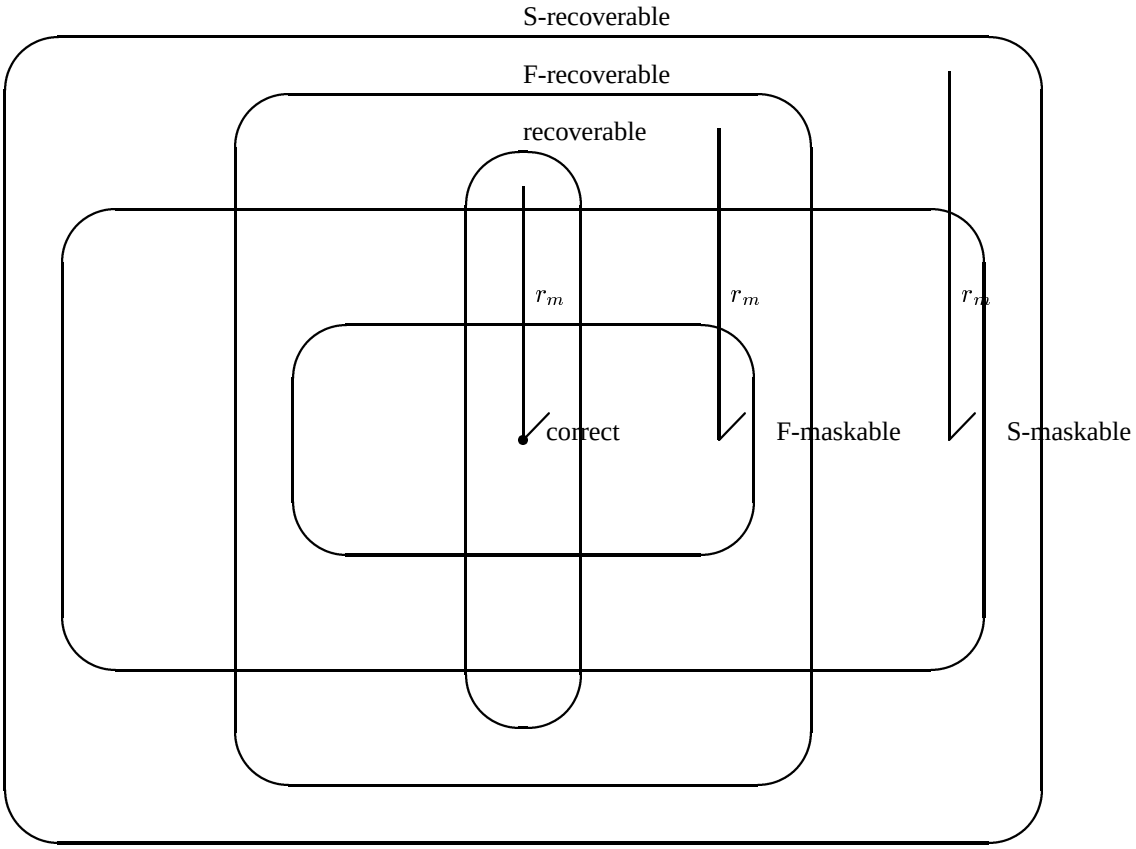


Figure 6: Correctness, Maskability and Recoverability

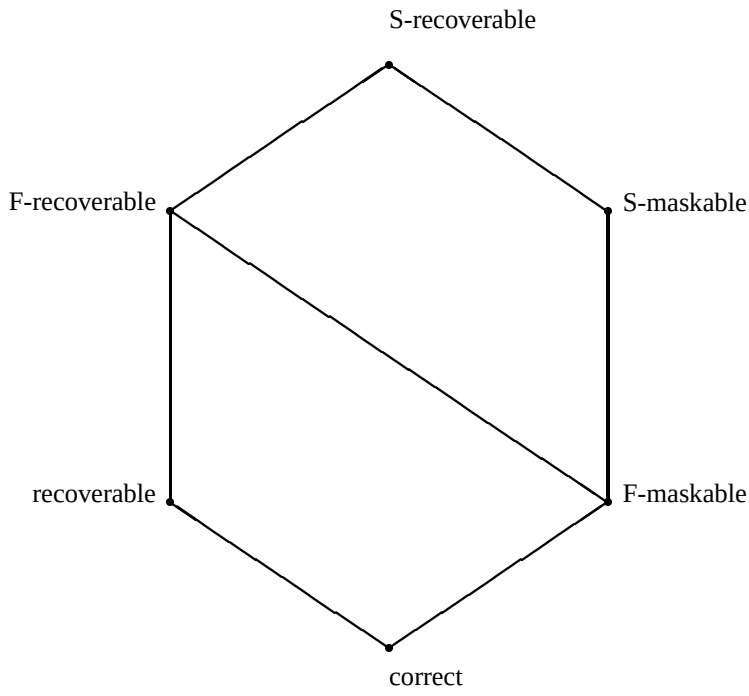


Figure 7: Hierarchy of Correctness, Maskability and Recoverability

## 2.3 Modeling Faults

We consider in turn two models for faults: a logical model, which discusses what a fault is and what fault removal consists of; and a statistical model, which analyzes the statistical laws that govern the distribution of faults in hardware and software systems, as well as the laws that govern fault sensitization.

### 2.3.1 Logical Modeling

While *failure* and *error* characterize the state of a computation, *fault* characterizes the function that a software or hardware product computes. We consider a program $P$ and a milestone $m$ on $P$. We let $\Pi_P^m$ be the past function of $P$ at milestone $m$ and $\Delta_P^m$ be the intended past function of $P$ at milestone $m$.

**Definition 16** *The past path of program $P$ at milestone $m$ is said to have a* fault *if and only if* $\Pi_P^m \neq \Delta_P^m$.

The first observation to make about faults is that while failure depends on the system's intended function, or the system's specification, faults are dependent on the system's *design*. What this means is that a fault is a fault *only* with respect to the designer's intent. This raises the following difficulty: in order to identify a fault we need, in principle, to have some indication of the designer's intent. We consider again the `modthreeofsquare` program:

```
program modthreeofsquare;
   begin
   read(s);
   s:= 2*s;   {<------ fault}
   s:= s mod 3;
   write(s)
   end.
```

If we identify the second line as a fault, it is only because we are assuming that the designer meant to compute the square of $s$, then the remainder by three of the result. It is conceivable (although not plausible in this special case) that actually the designer had meant to first double the input value, then take the remainder by 3 of the square of $\frac{s}{2}$ —for whatever reason. This means that the fault is the third line rather than the second.

```
intended program          actual program
begin                     begin
read(s);                  read(s);
s:= 2*s;                  s:= 2*s;
s:= (s div 2)**2 mod 3;   s:= s mod 3;   {<------ fault}
write(s);                 write(s)
end.                      end.
```

It is also conceivable (although again highly unlikely) that the designer intended to first double the argument, then raise it to the power 2 and divide the result by 4, then write the remainder by 3 of the result.

```
intended program          actual program
begin                     begin
read(s);                  read(s);
s:= 2*s;                  s:= 2*s;
s:= s**2 div 4;           s:= s mod 3;   {<------ fault}
write(s mod 3)            write(s)       {<------ fault}
end.                      end.
```

The main thesis of this (contrived) example is that the definition of a fault is very fragile, since neither the number nor the exact location of faults can be defined precisely. More generally, it is not difficult to imagine a situation where a fault can be fixed at two different locations in the program, or a situation where a fault can be fixed in many different ways, each involving a different number of changes. It is important to recognize that a fault is a characteristic of the *function* of a program or program part; if the function differs from the designer's intended function for that program or program part, then for some input value the actual function and the intended function will produce different outputs (causing an error and possibly a failure). It is not generally possible, for a given fault, to associate a statement or set of statements which, if modified in the proper way, will avoid the error/ failure caused by the fault. Hence there is no well-defined correspondence between faults (which characterize program functions) and individual faulty statements.

The vast majority of software faults stem from the product's design: the function defined by the software product does not satisfy the specification for which it was designed and built. It is possible to take into consideration *hardware generated software faults*, which are faults that stem from hardware failure but affect the function of a software product. Assume, for example, that upon storing a program $P$ in memory, some instruction was altered (by data transfer or data storage) and that the alteration was not detected. The function of the program is (potentially) different from the intended function —but the program itself is not questioned, as it is not responsible for the observed behavior. Consider the case where a hardware error, such as a flipped bit in memory, may change the conditions checked by the software, causing the software to exhibit an error. Another case may be that the software fails because it has not been designed to handle an unexpected hardware condition. Empirical studies [38] show that a majority of hardware errors that provoke software errors tend to be errors related to the networking hardware.

### 2.3.2 Statistical Modeling

In order to study fault tolerance, one must understand the nature of faults, their sources and manifestations. Two basic types of faults are [2]:

- Physical faults, and
- Human faults.

Physical faults stem from the physical phenomena internal to the system (aging, short circuits, etc.), or external (environmental, electro-magnetic, etc.). Human faults may be either design faults (requirements inaccuracies, design decisions and modifications) or interaction faults (violations of operating or maintenance procedures).

In the literature concerned with the assessment and evaluation of fault tolerant systems, the goal is to analyze the manifestations of faults at the system level (including hardware and software) using predefined knowledge about possible faults. The purpose of statistical modeling is to predict system behavior based on the known types and observed frequencies of faults. In other words, the term "statistical modeling of faults" includes two distinct activities:

- Collection and analysis of the type and frequency of faults that (may) cause failures, and
- Development of statistical models that, based on fault types and frequencies, predict the occurrence rates of failures.

The collection and analysis of fault types and frequencies is a necessary precondition for process and product improvement. In the remainder of this section, our interest is limited to modeling techniques used for the prediction of failure occurrence rates. Generally, the mechanisms guiding the sensitization of a fault into an error and, eventually, its propagation into a failure are complex. The number of system components and the complexity of interactions between them preclude a simple modeling of fault sensitization and propagation.

In statistical modeling, the most commonly desired quantitative system characterization is *the relative frequency of fault manifestations*. The frequency of fault manifestations may be represented by probabilities and averages. Typical point measures of fault manifestation frequency are the *mean time to failure (MTTF)* and the *mean time between failures (MTBF)*. In order to predict the behavior of the system, the failure probabilities must be determined as a function of time, forming probability distributions.

The *exponential distribution* is one of the most commonly encountered models used to describe the occurrence of hardware faults and their manifestations. The probability density function $f(t)$ (also called a *failure density function*) of the exponential distribution is expressed as:

$$f(t) = \lambda\, e^{-\lambda t},$$

where parameter $\lambda$ is the rate at which failures occur in time. The probability of encountering a failure by time $t$ is then

$$F(t) = P(0 \leq T \leq t) = \int_0^t f(x)dx \;=\; 1 - e^{-\lambda t},$$

assuming that the system starts its operation at time $0$. In reality, $\lambda$ is a function of time depicted by a bathtub shaped curve with three distinguishable phases: the infant mortality phase (high but decreasing value of $\lambda$), operational life phase (low and approximately constant value of $\lambda$) and wear-out phase (increasing values of $\lambda$). Thus, the constant failure rate, as used in the exponential distribution, is an approximation of the failure rate observed during the operational phase.

The general form of the failure density function given by *Weibull distribution* [33] is:

$$f(t) = \alpha\lambda(\lambda t)^{\alpha-1}e^{(-\lambda t)^{\alpha}},$$

where $\alpha$ is the shape parameter, and $\lambda$ is the scale parameter. The failure rate is a function $z(t)$, rather than a constant (in the case of exponential distribution), given by

$$z(t) = \alpha\lambda(\lambda t)^{\alpha-1}.$$

In studies of system reliability, $z(t)$ is frequently called a *hazard rate*, *hazard function* or *failure rate function*. The units of hazard rate are failures per unit of time. The additional flexibility provided by the Weibull distribution allows modeling of all the phases of the bathtub curve through the adjustments to parameter $\alpha$:

- If $\alpha < 1$, the failure rate is decreasing with time.
- If $\alpha = 1$, the failure rate is constant with time (resulting in an exponential distribution).

- If $\alpha > 1$, the failure rate is increasing with time.

If the time is considered being a discrete random variable, the appropriate probability distributions are the *Geometric distribution* (obtained by replacing $e^{-\lambda}$ with the discrete probability $q$), and the *Discrete Weibull distribution*. The questions which distribution should be used for modeling specific system (or a component) and how to select the parameters of the chosen distribution are addressed through the techniques for matching sampled fault/failure data to distribution models. An overview of these techniques (maximum likelihood estimators, linear regression analysis, confidence intervals, goodness-of-fit tests, etc.) in the context of fault modeling can be found in [33].

After the representative failure density function has been chosen for a given system, obtaining quantitative point estimates of fault manifestations is straightforward. The MTTF, for example, is the expected time that the system will operate before the first failure occurs. So, it can be computed as

$$MTTF = \int_{-\infty}^{\infty} t\, f(t)\, dt,$$

or, assuming that the system operation starts at time $0$,

$$MTTF = \int_{0}^{\infty} t\, f(t)\, dt,$$

where $f(t)$ stands for the failure density function. The probabilistic modeling techniques described so far have been successfully used in the context of hardware reliability, either at the level of a single component or at the system level. However, their straightforward adaptation to analyze the occurrence rates of software faults and failures was not nearly as successful.

Generally, the major source of software faults are human faults (requirements inaccuracies, design decisions and modifications, violations of operating or maintenance procedures). Therefore, simplistic assumptions such as constant failure rates and independence of events cannot be justified. As the consequence of the perpetual changes made to software components and systems (modifications, configuration management, builds, versions, etc.), the modeling of fault manifestations becomes very difficult, since the state of stability is rarely achieved. Predictive models relating software failure rate $\lambda$ to software fault content $\omega$ have been mostly abandoned. For example, a study by John Musa et. al. [25] suggests that $\lambda$ can be obtained from the following formula:

$$\lambda = \omega\ K\ r/I,$$

where $K$ is the fault exposure ratio (given the default value of $4.2 \times 10^{-7}$), $r$ is the processor speed (expressed in the number of instructions per second), and $I$ is the number of object instructions in the program. This and similar models have never achieved maturity and even their own authors do not advocate their use, due to the difficulties in parameter estimation and insufficient accuracy of failure rate predictions [24].

Another way to consider the relationship between software faults and failures is by studying the impact that faults have on the *input domain* of a program. The input domain is defined as the set of all the inputs relevant to a program. Typical input domains are very large, because they are formed as (a subset of) the Cartesian product of all possible values of all input variables (including the state variables within the program). Undoubtedly, some of the elements in the input space will reveal program faults, because these inputs will trigger errors and, eventually, failures caused by faulty program statements. The following derivation presents the simplified model of the relationship between software faults and failures, and it assumes that the program input domain is finite.

The probability $P_{NF}$ that a program will not fail is given by:

$$P_{NF} = \sum_{i=0}^{N-1} P(NF|i)P(i)$$

where $P(NF|i)$ is the conditional probability of no failure given that input $i$ is selected, $P(i)$ is the probability of selecting $i$, and $N$ is the total number of points in the input space. Note that $P(i)$ is, in fact, the same as the operational distribution function $f(i)$, which approximates the usage of the program within its operational environment. Now,

$$P(NF|i) = P(NF|i, i_{NF})P(i_{NF}|i)\ +\ P(NF|i, i_F)P(i_F|i)$$

where $i_{NF}$ and $i_F$ mean that the test case $i$ reveals a fault (faulty test case) or does not reveal a fault, respectively. Since in our simplified testing model the probability of failure is 1 if a faulty test case is selected and 0 for a non-faulty test case,

$$P(NF|i) = 1 \times P(i_{NF}|i) \ + \ 0 \times P(i_F|i).$$

The probability that $i$ is faulty does not depend on whether it is selected or not, so $P(NF|i) = P(i_{NF})$. Hence,

$$P_{NF} = \sum_{i=0}^{N-1} P(i_{NF}) \times f(i).$$

If the probability that a test case $i$ is faulty has the same value for all $i$, representing a uniform random distribution of faults in the input space, $P(i_{NF}) = g$, then,

$$P_{NF} = \sum_{i=0}^{N-1} g \times f(i) = g \times \sum_{i=0}^{N-1} f(i) = g \times 1 = g.$$

Under the assumptions of the above derivation, the probability of failure (and, consequently, the failure rate) for a program can be represented by a constant. This conclusion is very specific and in most cases misleading. First of all, $P_{NF}$ is derived from exhaustive testing (the sum over all the $N$ elements of the input space). In practice, exhaustive testing is impractical, thus the quality of the estimate critically depends on the choice of tests, i.e., the representativeness of the *operational distribution* $f(i)$. In other words, the failure rate of a program is "correlated to how frequently the design faults are exercised as opposed to the number of faults present" [33].

Yet another assumption in the derivation above does not occur commonly. There is some evidence to the effect that faults are not distributed uniformly throughout the input domain. Finelly and Bishop provide experimental and theoretical evidence that faults are grouped into contiguous regions of the input domain (called 'blob' effects [5, 4] and error crystals [12]). Empirical data gathered by Hecht and Crane [13, 14] lead to the conclusion that code segments which are rarely executed have a much higher failure rate than frequently executed code. Also, failures may occur as the consequence of the interaction between two or more faults, spawned by specific system conditions.

An alternative approach to statistical modeling of software faults is based on software metrics. It has been shown that the pattern of faults in software (or a component) is related to its measurable attributes, static as well as dynamic. Generally, if a program module is measured and found to be complex, then it will likely have a larger number of faults. Typical fault prediction techniques of this type take software complexity metrics as independent variables, and software quality measures related to faults as the dependent variables. The models based on multivariate statistical techniques such as multiple discriminant analysis and multiple regression analysis, have demonstrated a promising level of accuracy in predicting fault content of software modules. This is of great help in software development process, since it allows the developers to concentrate their improvement efforts to the group of modules with the highest predicted fault content. A potential advantage of the metrics based approach is that the program failure rate prediction can benefit from the measurement of dynamic complexity. The dynamic complexity of the program will change in response to the varying inputs. Given the association between the module complexity and faults, it follows that as an application changes over time, so too will the likelihood of failures change with time [23]. Currently, additional research including experimental data and metrics collection and the investigation of fault-failure triggering and propagation mechanisms, is needed to improve the state of the art in statistical modeling of software faults and failures.

## 2.4 Illustrative Examples

### 2.4.1 A Hardware Example: The Pentium ™Processor FDIV fault

On October 30th, 1994, Dr. Thomas R. Nicely of Lynchburg College posted a message to CompuServe describing an unexpected result he had found in one of his experiments involving prime number calculations on a PC with an Intel Pentium™processor. In June 1994, in the course of intensive number crunching, Dr. Nicely realized that the answer to the division of 1 by 824633702441 was accurate to only eight of the fifteen decimal places. He had first noted the

problem in June, then excluded all other sources of error, and he reported it to Intel on October 16th. At the end of November, Intel released a report [31] describing the nature of the hardware fault. Shortly thereafter Intel released a software patch [9] (as a short-term solution) and finally adopted a policy for replacing the faulty CPUs (as a long-term solution). Because the fault occurs in certain cases when a hardware division (FDIV) is performed, it is commonly known as the *Pentium FDIV bug*.

### The fault

The fault that is the cause of the Pentium FDIV bug is in effect the omission of five entries in a hardware PLA (Programmable Logic Array) used in the hardware implementation algorithm for the divide operation [31].

### The error

The fault manifests itself as an imprecise result from a divide operation. This can occur whenever one or more of the five missing entries are looked up in the hardware divide algorithm. According to Intel [31], the inaccuracy occurs in the 4th to 19th significant decimal digits of the division result. The following example illustrates how the error would be sensitized [7] using a calculator or spreadsheet running on a Pentium-based PC:

Fault-free Pentium: 5505001/294911=18.66665197
Faulty Pentium: 5505001/294911=18.66600093

### The failure

Because the worst-case inaccuracies occur in roughly the 4th significant decimal digit of the result, a failure occurs whenever

- the error has been sensitized and,
- the inaccurate digits (at the 4th to 15th decimal places) of the result are used in a subsequent calculation.

The following example illustrates the gravity of such a failure. Consider the calculation of the equation $A - ((A/B) \times B)$, which in fact simplifies to $A - A$, or 0. Using the values of $A = 5505001$ and $B = 294911$ as in the example shown above, a spreadsheet calculation of the equation on a faulty Pentium would would yield a result of 192 rather than 0.

### The Follow Up: Fault Removal

Intel's software patch to the Pentium bug essentially causes the division unit to avoid looking up the five faulty (missing) entries in the PLA. If the divisor has been identified as a value that can provoke the imprecision fault, the algorithm then scales both the numerator and divisor equally, and performs a "safe" divide. This fault tolerance technique can be viewed as temporal redundancy (as the algorithm will take more time) as well as spatial/informational redundancy, since there are still valid entries in the PLA that can be used to effectively perform a division (different forms of redundancy will be discussed in section 4).

The Pentium FDIV bug is clearly a design fault in the processor —important information was omitted from a table used in an algorithm to perform divisions. Because the divide algorithm implemented in hardware does not behave appropriately, this fault can also be viewed as a software fault triggered by incorrect data in the hardware. One can imagine that the hardware PLA could contain faulty information that was not due to design faults but rather to physical wear or environmental conditions (flipped bits in the PLA). In this case the division algorithm that relies upon the integrity of the table's data would still be susceptible to the faults. An algorithm such as the one proposed by Intel could be used to effectively perform two divisions —one using the normal numerator and divisor, the other using using the scaled values in order to make use of distinct lookups in the PLA. The results could be compared to detect faults. Similarly, faults could be detected in the division unit using informational redundancy with the equation $A - ((A/B) \times B) = 0$.

#### 2.4.2 A Software Example: The Ariane 5 Exception Handling Fault

On June 4, 1996, the maiden flight of the *Ariane 5* European Space Agency (ESA) launcher ended in a failure. The investigation of the failure started immediately, and its findings were made public a few weeks later [28]. The Flight

Control System of Ariane 5 is designed to receive measurements of the attitude of the launcher and its movements in space from an Inertial Reference System (SRI). SRI has its own inertial computer in which angles and velocities are calculated on the basis of information from a "strap-down" inertial platform, with laser gyros and accelerometers. The data from the SRI are transmitted through the data bus to the On-Board Computer (OBC) which executes the flight program and controls the nozzles of the solid boosters and Vulcain main engine via servo-valves and hydraulic actuators.

System level fault tolerance is achieved through hardware redundancy. Two SRIs (SRI-1 and SRI-2), equivalent in hardware and software, operate in parallel as an active primary component and a "hot" stand-by secondary component. Since the design of SRIs on Ariane 5 is the same as that used on Ariane 4 (an earlier model of the launcher), project management decided to reuse the SRI software without additional testing. This decision was made in spite of the fact that the early part of the trajectory of Ariane 5 differs from that of Ariane 4 and results in considerably higher values from sensor readings.

### The fault

Approximately 39 seconds into the flight an Operand Error appeared when SRI executed a data conversion from 64-bit floating point to 16-bit unsigned integer value called BH, Horizontal Bias, related to the horizontal velocity sensed by the platform. This computation was required on Ariane 4 but meaningless on Ariane 5, remaining in the code due to software reuse combined with the lack of funds for component modification. The On-Board Computer (OBC) correctly recognized the failure diagnostic bit pattern coming from SRI-1 and switched the control to the "hot" spare SRI-2.

### The error

In the next computational cycle, 72 milliseconds later, the same failure diagnostic bit pattern was communicated to the OBC by SRI-2. Since OBC could not switch the back-up again because SRI-1 had already declared a failure, it wrongly interpreted the diagnostic bit pattern as proper flight data and commanded nozzle deflection.

### The failure

Fully deflected nozzles of the solid boosters and the Vulcain main engine resulted in an angle of attack of more than 20 degrees. Boosters were, in turn, separated from the main stage, triggering the self-destruct system of the launcher. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 meters, the launcher veered off its flight path, broke up and exploded [30].

### The Follow Up: Post Mortem

The Inquiry Board's findings indicate that the project management jointly agreed not to include the Ariane 5 trajectory data in the SRI requirements specification. The Operand Error in conjunction with the specification of exception-handling mechanism caused the mission to fail. The decision to shut down the SRI processor operation after it reported a failure proved to be fatal, because its restart was not feasible since altitude is too difficult to re-calculate. According to the Inquiry Board, the reason behind this drastic action (shutdown) lies in the culture within the Ariane program of only addressing random hardware failures, while the potential consequences of software faults were ignored [28].

# 3 Deploying Fault Tolerance

## 3.1 Error Detection

In its strictest sense, error detection means determining whether the current state is strictly correct at the current milestone. While for the purposes of error recovery the relevant test is recoverability, for the purposes of fault removal the relevant test is strict correctness of the current state, in the sense of definition 10. As soon as we find that the current state is not strictly correct, we infer that we have a fault to remove. Error detection poses two difficulties in practice, which are:

- In order to check that the current state is strictly correct, we need a copy of the initial state; in practice, it is not always possible or practical to keep a copy of the initial state (which can be as complex as the content of many

files, or a specific configuration of sensor data). Even when it is possible, this requirement imposes a potentially significant space penalty.

- In order to check that the current state is strictly correct, we need some (executable) representation of the past function at the current milestone. This requirements imposes several penalties: a development cost penalty, in terms of producing an alternative means to compute the past function and check its result against the current state; a run-time penalty, in terms of duplicating the execution of the past function, and checking for equality or equivalence; a reliability penalty, to the extent that there are cases when the actual past path executed properly but the alternate past path didn't, hence creating a false alarm.

Because of all these complications, it is rather seldom that error detection is performed by testing the condition $s = \Delta_P^m(s_0)$. Rather, we attempt to test weaker versions thereof (see figure 8).

- Because it is too much trouble to keep a copy of $s_0$ and recompute the function $\Delta_P^m(s_0)$, we may instead test the current state to check, not that it is the image of the exact initial state $s_0$, but that it is the image by $\Delta_P^m$ of *some* initial state. This amounts to checking that the current state is in the range of function $\Delta_P^m$; in other words,

  $$s \in \mathit{rng}(\Delta_P^m).$$

  In terms of figure 8, this solution consists in checking that the current state lies inside the inner oval. It is very common, in software fault tolerance, to adopt this solution as a compromise for error detection: it spares us the space overhead of having to store $s_0$ and the time overhead of having to recompute $\Delta_P^m(s_0)$; in exchange, it runs the risk of missing errors where the current state is not the image of the initial state by the past function, but is nevertheless in the range of the past function.

- There are cases when even checking that the current state is in the range of the past function is too much overhead. In hardware, we perform error detection by defining a space in which the current state must lie: this space is defined by a redundant coding of data (e.g. by adding a parity bit). Whereas the test discussed above is independent of $s_0$ but still depends on function $\Delta_P^m$, the current test is independent of both $s_0$ and $\Delta_P^m$: the same test (e.g. for parity bit) is performed at all milestones. In terms of figure 8, this solution consists in checking that the current state lies inside the outer oval. Again, this test represents a compromise with respect to the test discussed above: it spares us the trouble of having to derive a different test condition for each milestone —the same circuit will test all milestones; in exchange, a data value may well be outside the range of the past function yet fail to be detected as it is still within the space (defined by e.g. the parity bit protocol).

## 3.2 Damage Assessment

We have mentioned in section 2.2.4 that damage assessment consists, in effect, in analyzing the current state to determine where does it lie with respect to the boundaries drawn in figure 6. To fix our ideas, we take two simplifying hypotheses:

- We are interested in failure freedom with respect to the expected function of the system, rather than a predefined specification thereof; this rules out S-maskability and S-recoverability from further consideration.

- We disregard the concept of strict *recoverability*, because it is not necessary to retrieve the exact correct state at milestone $m$ —it suffices to retrieve a maskable state.

Following these assumptions, we are left with the following concepts: correctness, F-maskability, and F-recoverability. Figure 9 reflects the hierarchy between these levels of correctness. Damage assessment then consists of asking whether the current state is within the inner oval; if it is not, we then ask whether it is within the outer oval.

- If damage assessment is invoked at all, it is because the current state is not correct; hence we are not at the center of figure 9.

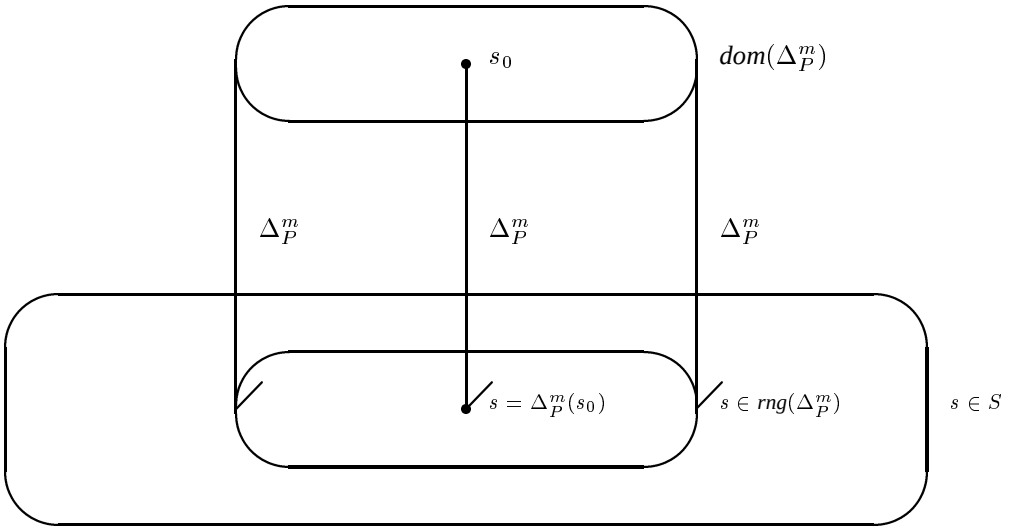


Figure 8: Error Detection Options

- If the state is within the inner oval, then it is F-maskable; no recovery is necessary —we may return control to the future function, which will (barring further problems) produce a failure-free final state.
- If the state is outside the inner oval but inside the outer oval, then the state is deemed non F-maskable, but still F-recoverable. Hence we must invoke a recovery routine then resume the computation.
- If the state is outside the outer oval, then it no longer contains sufficient information to ensure failure free execution; the only way to salvage the computation is to retrieve a maskable state from some source (e.g. backward recovery).

Damage assessment provides a framework for error recovery, which we discuss in the next subsection.

## 3.3 Error Recovery

Error recovery is the process of regaining operational status and restoring a system's integrity after the occurrence of an error. Some form of redundancy (sec. 4) must be exploited to recover from an error. Error recovery can be thought of as either forward recovery (in which the current state is mapped into a maskable state) or backward recovery (in which a prior error-free system state is restored). Forward recovery techniques for hardware include N-modular redundancy (NMR) with voting, and error-correcting codes. Backward recovery techniques for hardware typically involve retrying the faulty operation, on the assumption that the fault is transient, hence will not cause an error in subsequent attempts (e.g., after an error from a disk controller).

In the context of software fault tolerance, error recovery is invoked in the course of a damage assessment routine, whose structure is outlined as follows:

```
procedure errorrecovery;
begin
if not correct then
   begin
   errorreport; {signal error}
   if not maskable then
```

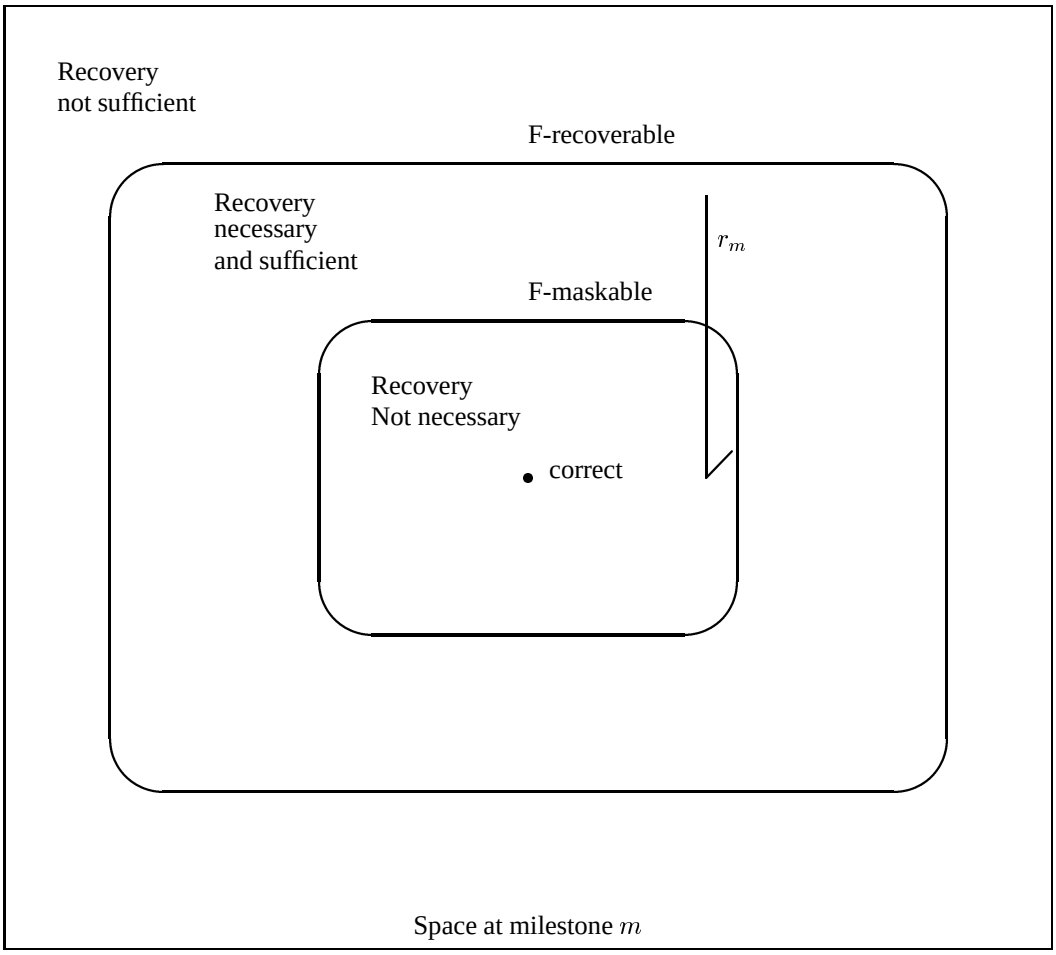


Figure 9: A Framework for Damage Assessment

```
        {recovery is necessary}
        if recoverable then
           {recovery is sufficient}
           forwardrecovery
        else
           {recovery is insufficient}
           backwardrecovery
     else
        {no intervention is required}
     end;
  end.
```

Of course, it is not possible to give a general characterization of the various procedures that arise in this segment of code (`correct`, `maskable`, `recoverable`, `forwardrecovery`, `backwardrecovery`); in [21], Mili gives characterizations of some of these under special conditions. Generally speaking, forward recovery is applicable to states that are not maskable but still have all the necessary information to lead to a failure-free final state; such states are mapped into a maskable state and the execution resumes. Backward error recovery is applicable to all states since in fact it does not refer to the current state; instead it retrieves a state that is known to be correct (possibly at an earlier milestone), transforms it into a state that is (at least) maskable at the current milestone, and resumes execution. Backward error recovery offers the advantage that it is generally applicable, but is very costly in terms of memory (duplication the program state) and in terms of execution time (overhead of saving the state at regular intervals). The most generic pattern of backward error recovery is the *recovery block* [26, 27].

In practice, it is not always necessary to perform such a detailed analysis of the current state for the purpose of damage assessment. Because a maskable state is *a fortiori* recoverable (with a recovery routine $r$ that preserves the state) we can invoke the recovery routine whenever the state is recoverable, whether or not it is in fact maskable. In addition, we may or may not have provisions for backward error recovery; in case we dont, all we can do when we find a state that is not recoverable is to report a failure. In such cases, the error recovery procedure can be articulated as follows:

```
procedure simplifiederrorrecovery;
begin
if not maskable then
   if recoverable then forwardrecovery
   else failurereport
end.
```

## 3.4 Fault Removal

Traditionally, fault removal is defined as the activity aiming to improve system reliability (as well as the other quality attributes) through detection and elimination of faults in the later stages of the development life cycle. In this section, however, we use a much narrower definition of fault removal, suitable for our study of hardware and software fault tolerance. We define fault removal as the activity aiming to improve system reliability through the elimination of faults during the operation of a fault tolerant system. Consequently, a fault removal technique works in concert with the error recovery mechanism, thus helping to prolong the operational life of the system. Within the boundaries of the above definition, basic differences between fault removal techniques for predominantly hardware as opposed to predominantly software fault tolerant systems stem from the origins of faults in hardware and software.

In case of hardware components the nature of the physical design process coupled with the rigidity of component boundaries, inability to implement frequent changes, and the generic nature of components, are some of the reasons that most of the observed faults and failures during the operational life fall under the category of wear and tear. When compared with software, hardware components are moderately complex designs. For these reasons, releasing nearly zero defect hardware is a reality.

In case of software, the situation is just the opposite. Faults and failures observed during the operational life mostly originate in the requirements and design phases of the life cycle. Achieving “perfection” through fault avoidance and

removal techniques is a tedious and costly process, which can be justified only for high-consequence systems [18, 43]. Furthermore, the "perfection" is never guaranteed, leaving developers and users the option to rely on the statistical confidence argument.

In the case of a fault tolerant system based on modular redundancy fault removal capability is enabled through *standby sparing* technique [16]. If the error detection system reports existence of a permanent fault, system reconfiguration is initiated. Reconfiguration consists of the replacement of a faulty (primary) component by an identical component from the pool of spares. Reconfiguration is not an instant operation. If the disruption in processing must be minimized, *hot standby sparing* is used. In hot standby sparing, spares operate in synchrony with the primary modules. In such a way, the state information in the hot spare and the primary component is identical, and the spare is able to take over quickly at any time. In the *cold standby sparing* technique, spares are unpowered until there is a need to replace a faulty primary module. When the efficient energy consumption requirement is more critical than the timeliness requirement, cold standby sparing is preferred.

The use of standby sparing technique for software fault tolerance is limited to systems that operate in hostile environments. For example, the operation of space-borne systems is often affected by radiation, causing single event upsets. Random bits in computer registers and memory cells can be flipped with devastating effect on program logic control. Assuming that a spare software component is has not been affected by the same upset, it can be used to replace the faulty process [8].

Software fault removal in an operational fault tolerant system is a novel category, that has been investigated recently in the context of it on-board maintenance for new generation space exploration missions [37]. On-board preventive maintenance generally refers to the actions that take place during a mission aimed at eliminating or minimizing potential error conditions that accrue over the operational life of a system. *Software rejuvenation* techniques [15] address the aging phenomena such as memory leakage and data corruption.

These phenomenon can be tackled via program reinitialization which cleans up the systems internal states (complete age reversal). The related technique, falling under the category of perfective maintenance of space-borne systems, is *software upgradeability* [1]. It allows that a particular software module (instead of the entire flight software) can be uploaded without interrupting system operations. Finally, the corrective maintenance phase addresses the traditional notion of fault removal. It occurs after the system detects that it is in an erroneous state. Since an upgraded software component may not be instantly available, a backup *approximate computation* can be employed for forward recovery [37].

# 4 Design of Fault Tolerant Systems

## 4.1 Redundancy as the Basis of Fault Tolerance

Fault tolerance in any system is achieved through redundancy in some form or another. Redundancy can be classified in several ways. One such classification identifies three kinds of redundancy: temporal redundancy, spatial redundancy, and informational redundancy. In this section we consider in turn these three forms of redundancy, and discuss how they can be deployed in hardware and/or software to achieve fault tolerance. Also, we consider how provisions of redundancy must be balanced against other design criteria pertaining to the performance, cost, and efficiency of a fault-tolerant system.

It is instructive to observe how redundancy, in its different forms, affects the different phases of fault tolerance.

- *Temporal Redundancy*. Most generally, temporal redundancy is used for the purposes of error recovery; also, in extreme cases, it may be used for the purposes of error detection (see section 3.1).
- *Spatial Redundancy*. Spatial redundancy is used for the purpose of error detection, fault removal (identification) and error recovery.
- *Informational Redundancy*. Informational redundancy is used for error detection, damage assessment, and error recovery.

These two-dimensional relationships between forms of redundancy and phases of fault tolerance are summarized in figure 10.

| Types of Redundancy | Principle | Phases of Fault Tolerance | | | |
|---|---|---|---|---|---|
| | | Error Detection | Fault Removal | Damage Assessment | Error Recovery |
| Temporal Redundancy | Duplicating Computations | √ | | | √ |
| Spatial Redundancy | Duplicating Components | √ | √ | | √ |
| Informational Redundancy | Duplicating Data | √ | | √ | √ |

Figure 10: Forms of Redundancy vs Phases of Fault Tolerance.

## 4.2 Operating Cost vs. Temporal Redundancy

Temporal redundancy achieves fault tolerance by duplicating computations: it is used to detect errors (by executing some block of code, then executing a test to check its validity), and to recover from errors (by making provisions to retrieve a correct state and resume the computation therefrom). Typically, this involves some time overhead to recover from the error, perhaps by retrying the operation that produced the error. Examples of temporal redundancy in hardware include retries of I/O operations: A disk controller will automatically retry a disk read whenever it detects a bit parity error; and a bus will retry a transfer of information whenever it detects a message format error. Such cases are straightforward, for two reasons: first, even though the operation is erroneous, it has caused no loss of crucial information —hence no complex error recovery is required; second, such provisions are made under the hypothesis of transient faults, and are dealt with by merely re-executing a predefined sequence of statements —hence no complex fault removal is required. Instances of hardware fault tolerance resulting from temporal redundancy and involving elaborate mechanisms for on-line fault removal include such techniques as disabling a faulty component and enabling a fault-free standby component.

In software, temporal redundancy can include simple retries just as in hardware. However, there are also some software-specific techniques such as *checkpointing*, *journaling* and *recovery blocks* that invoke temporal redundancy. In checkpointing, a portion of the system's state is saved at certain checkpoints during the execution of the software. If an error occurs, the system state is restored to the most recent checkpoint, and the software execution is resumed. In this example of temporal redundancy, the tradeoff on performance is evidenced by the additional time required for the system to make the checkpoints, as well as the time required to re-execute the software instructions between the checkpoint and the point at which the error is detected and handled. As the distance in time increases between checkpoints, so does the recovery time. Yet as the frequency of checkpoints increases, so does the time overhead due to the checkpoint operation. Journaling is the simplest of the techniques for temporal redundancy in software, but is the most costly in terms of recovery time. At the instantiation of the software, a copy of the initial state of the system is stored. As the software executes, all actions affecting the systems state are recorded. Then, if the software fails, recovery is possible by re-executing the recorded actions. The technique known as recovery blocks (RB) [27] combines elements of checkpointing and reconfiguration with alternate software modules (recovery blocks). Each recovery block contains state information pertinent only to that particular block and that will be automatically saved if altered within the block. The primary alternate is first executed and an acceptance test is performed at the end of the execution of that block. If the test is not passed, then one of the other alternate blocks is executed using the checkpoint state information. This process is continued until the acceptance test is passed, or there are no remaining alternate blocks. The RB technique has the advantage that the checkpoint information is both minimized and made explicit: only the information within the scope of the block is saved.

Temporal redundancy impacts the overall performance of the system, and this can be important especially in systems with high loads or systems with real-time requirements. In terms of the overall performance of the system, a designer should consider the potential frequency of the occurrence of errors that will be handled with temporal

redundancy, as well as the additional time required for a retry when such errors do occur. The system must be able to accommodate the additional time required for recovery so that it does not provoke other faults in the system (e.g., missed deadlines). Because of its dynamic nature, temporal redundancy lends itself more to an implementation in software than in hardware, which is evidenced by the various examples above. On the other hand, because software is generally more complex than hardware, it may be more difficult for the system designer to predict the recovery time needed in a temporally redundant software implementation. The time needed for temporal redundancy in hardware tends to be more predictable.

## 4.3 Development Cost vs Spatial Redundancy

Spatial redundancy involves the duplication of hardware or software components to provide for error detection (by comparing the results delivered by alternate components), fault removal (by voting out fault components) and error recovery (by adopting the results produced by fault free components). The deployment of spatial redundancy depends on two considerations, which we analyze in turn; in light of this analysis, we will discuss how spatial redundancy is deployed in hardware, and in software. The first consideration that determines how spatial redundancy is deployed is fault modeling.

- *Transient or Incidental Faults*. There are cases when the faults that are most likely to occur are of a transient nature, or are triggered by environmental conditions that are external to the component. This is generally the case with hardware faults, or hardware-generated software faults.
- *Design Faults*. There are cases when the faults that are most likely to occur stem from design mistakes; these are permanent in nature and are sensitized by the right combination of inputs. Because software products are generally complex, they are prone to design faults; also, because software products typically deal with an infinite (or prohibitively large) input space, they are prone to have residual faults even after extensive testing and debugging.

The second consideration that determines how spatial redundancy is deployed is the cost of duplicating components.

- *Hardware costs*. Like most manufactured products, hardware components are produced in large quantities from a single design. Consequently, the cost of their design is a small fraction of their production cost, because it is amortized over the production volume. Hence the cost of a hardware component is due primarily to manufacturing costs.
- *Software costs*. Not only is the cost of designing software very high (the subject of long standing lore), the cost of manufacturing software is very low (barely more than the cost of a diskette, or a CD-Rom). Consequently, most of the cost of a software product is traced to design costs.

Because the faults that are most likely to occur in hardware components do not involve the component's design, it makes perfect sense to use several components from the same batch, even though they have identical designs. Because software faults are most typically design faults, we cannot deploy spatial redundancy in software by means of identical copies of the same product; rather, we must apply *design diversity*, even though this involves high development costs.

In hardware, the cost of a system using spatial redundancy increases by the degree of duplication and the cost of the duplicated components, as well as any additional components needed to effect the technique (voters, comparators, additional input and output lines, etc.). In software, there are higher costs due to limited design resources, and to concurrent execution of redundant programs. The cost for fault detection and masking with spatial redundancy in hardware can be lower than with other methods (e.g., self-checking circuits) because they are relatively simple to implement and useful at various levels of the system. Furthermore, spatial redundancy is used quite effectively to maximize the yield of VLSI fabrication by mitigating faults due to manufacturing defects that are common in VLSI production [20].

Apart from the additional cost, the disadvantages to spatial redundancy can be the potential for weak links that limit the overall reliability of the system. For example, in a hardware implementation of triple modular redundancy (TMR) with voting, the voter can be a single point of failure that can defeat the TMR scheme. Similarly in software N-version programming (NVP), the non-replicated software driver (the software that controls the execution of the

various software versions and performs the comparisons on the results) is a single point of failure. An advantage of spatial redundancy is that physical (non-design) faults can be detected easily using identical redundant modules. For example, two identical copies of a functional element are employed, and the result from each is compared. If a single fault has occurred in one of the elements, the two results will differ.

An excellent example of the practicality of spatial redundancy in hardware is *Redundant Array of (Independent) Inexpensive Disks* (RAID), a technique for grouping hard disk drives to improve performance and data availability. RAID devices are multiple disks that appear as one disk to the operating system. The salient feature of RAID is that the spatial redundancy can be used flexibly, depending on the requirements of the system. Performance can be improved by enabling simultaneous accesses to disk storage by a technique known as *striping*, or accessing multiple disks in parallel. Fault tolerance can be improved by storing redundant information on separate disks. Because of its flexibility, RAID systems have a range of configurations, from those which provide pure performance (no fault tolerance) to those which provide high integrity (with little or no performance enhancement), to those which provide a compromise between the two.

## 4.4 Design Complexity vs Informational Redundancy

While temporal redundancy duplicates computations and spatial redundancy duplicates components, informational redundancy duplicates *data*. This can be as brute force as creating multiple copies of the same data, or as subtle as coding information in a redundant manner. Informational redundancy can be used for the purpose of error detection (exploiting the redundancy to determine that some values are outside the predefined code) and error recovery (if the code is sparse enough, map illegal values onto the closest legal code).

The most common example of hardware-based informational redundancy is found in error-detecting codes and error-correcting codes. For hardware, a large proportion of modern primary memory designs use Hamming single-error correcting codes [33]. For software, informational redundancy can be used in a similar way, such as cyclic redundancy checks (CRC) on blocks of data either stored in memory, on a disk or transferred over a communication link. Voting techniques in both hardware and software are a form of informational redundancy that can be functionally equivalent to coding techniques.

Informational redundancy is exploited with the goal of maintaining the integrity of data in a computer. Even though the individual hardware and software components of a system may seem to be fairly reliable, because of the complexity of modern computers, faults can occur when the data is stored, transferred or manipulated. The coding of data with additional (redundant) information is how errors can be detected and even corrected. Coding of information is done to check/correct the resultant data from arithmetic operations (e.g., arithmetic codes), logical operations, and data transfers (e.g., communication coding). Examples of specific coding techniques employed in these respective cases are *AN Codes*, *duplication*, and *parity*.

Error detection usually results in an interruption of the normal operation, and the computational step that is believed to have caused the error can be retried, as described above for temporal redundancy. Error correction, however, allows the computational process to continue without interruption. Although error correction would seem to be advantageous over error detection, it is important to consider a few issues. First, if we consider a fixed amount of informational redundancy to be used for error correction, then as the number of potential error patterns to be corrected increases, the probability of an undetected error also increases (as well as the complexity of the coding computations) [26]. Therefore if, for example, temporal redundancy can be used effectively, then it may be more practical to employ simpler error detection codes (which often requires less informational redundancy than error correction codes). While hardware-based informational redundancy is artificial (built-in at design time for the specific purpose of fault tolerance), software-based informational redundancy is most often natural, stemming from the natural richness in the state of a program. Also, while hardware-based informational redundancy is consciously built into the design for the purposes of fault tolerance, software-based informational redundancy arises typically as a byproduct of the design, and can be exploited *a posteriori* to achieve fault tolerance. If we consider the error detection formula

$$s \in \mathit{rng}(\Delta_P^m)$$

discussed in section 3.1, we find that the only reason it is ever meaningful to check this condition is that program functions are typically very *non-surjective*: the range of values that are admissible at milestone $m$ is much smaller than

the range of values that can be represented by the program variables at that milestone. For example, after executing the statement `x:=x*x`, where $x$ is a real variable, we can check the condition $x \geq 0$; also, after executing the statement `y:= 2*x`, we can check the condition $y = 2 \times x$. By contrast, after executing the statement `x:= x+1`, there isn't much we can check about $x$. This is because the last function is surjective, whereas the first two are not. In programming terms, we can make program functions less surjective by introducing variables, hence creating the potential for more interrelationships between program variables —and enhancing the error detection capability.

It is also possible to create artificial informational redundancy in software, for the purpose of software fault tolerance. The archetypical example is the *recovery blocks* mechanism, whereby we assume that the program is divided into (recovery) blocks, which have the following generic form:

```
begin
sprime:= s;
body; {modifies s while preserving sprime}
if not maskable(sprime,s) then s:=recover(sprime)
end;
```

Variations on this generic pattern include: saving only those variables that are modified by `body`; providing for a sequence of increasingly weak executable assertions, and associating increasingly desperate recovery measures. The latter example is an instance of damage assessment, and justifies our claim that informational redundancy provides a basis for this phase (see figure 10).

# 5 Impact of Fault Tolerance on Reliability

The goal of redundancy techniques in Fault tolerant systems is to enhance the system reliability. It is claimed that system reliability can be increased considerably through hardware and software components redundancies. This claim has been borne out for hardware components on one hand, where redundancies in highly reliable hardware components have proved to be effective and warrants the added design complexity. Effectiveness of redundancies in software components, on the other hand, may be hampered by the existence of correlated failures. Experiments have shown that incidence of correlated failures in fault tolerant software may not be negligible in the context of current software development and testing techniques.

This section will discuss the reliability modeling techniques of hardware and software systems. Combinatorial and Markov model techniques for hardware systems will be discussed first. Data-domain modeling, and time-domain modeling of software reliability will be briefly discussed next. Then, techniques for analyzing the reliability of fault tolerant systems using measurements and simulations will also be discussed.

## 5.1 Hardware Reliability Modeling

Reliability is one of the most important attributes of fault-tolerant digital systems. This section introduces the basic concepts of hardware reliability modeling. We start first with a definition of this attribute then the modeling and evaluation techniques are briefly discussed.

**Definition 17** *The* reliability *of a hardware system (or a hardware component of the system) during a mission interval is defined as the conditional probability that the system (or the component) operates correctly throughout the mission interval given that it was operating correctly at the starting time instant.*

Suppose that a sample of $N$ identical components were tested by placing them in operation and then recording the number of failed components after a mission time $t$ as $N_f(t)$, then the reliability of this of components in this sample is given by,

$$R(t) = 1 - \frac{N_f(t)}{N}.$$

**Definition 18** *The normalized derivative of $N_f(t)$, with respect to the number operational components, defines the* instantaneous failure rate of components $z(t)$. *The units of the failure rate function are failures per unit of time.*

$$z(t) = \frac{1}{N - N_f(t)} \times \frac{dN_f(t)}{dt}.$$

From the above definitions, it can be easily shown that the relation between $R(t)$ and $z(t)$ is given by,

$$\frac{dR(t)}{dt} = -z(t)R(t).$$

The commonly accepted time behavior for z (t) is a bathtub shaped function, as has been discussed in section 2.3.2. Standards for evaluating the constant failure rate of hardware components have evolved for many years [35]. These standards provide a model for the failure rate of electronic components using experimental data obtained by analyzing the failures of actual devices.

### 5.1.1 Reliability Modeling Techniques

The most commonly used approaches for reliability modeling of hardware systems are combinatorial models and Markov models. The first approach uses probabilistic techniques that enumerate the possible independent events that lead to a system being operational. The probabilities of these events are determined and then used to form an estimate of the system reliability. The Markov model, on the other hand, consists of the possible system states and the possible state transitions. A system state is defined by a distinct combination of faulty and fault-free components of the system. State transitions represent failure or repair events of the system components. The reliability of the system is estimated from determining the probabilities of the operational states of the system.

### 5.1.2 Combinatorial Models

The combinatorial model generally derives the system reliability from the reliabilities of the system components. A reliability block diagram consisting of series or parallel configurations of components is constructed. This diagram can be thought of as a flow diagram, from the inputs to the outputs of the system. A series configuration consists of those components that must remain operational for the system to operate. In a parallel configuration, on the other hand, only one of the components must be operational. The system reliability for a series configuration of components is obtained by multiplying the reliabilities of the components, assuming independent fault behavior. The system reliability for a parallel configuration, which represents redundant components in a fault tolerant system, is evaluated from the component reliabilities by calculating the probability of having at least one component operational. Parallel configurations in a reliability block diagram are reduced to equivalent single elements with the same reliability. The diagram reduces to a series diagram where the overall system reliability can be easily evaluated.

### 5.1.3 Markov Models

The reliability block diagram can be extremely difficult to construct for complex systems. In addition, the fault coverage factor and the repair process can be very difficult to model in combinatorial fashion. Markov models [40] have been used to incorporate the above important concepts in the system reliability model. The coverage factor is easily represented by a transition modeling the event of imperfect coverage. The repair process is modeled in a similar way as the failure process by adding transitions representing repair activities. Components repair rates are estimated and used in the model in a similar way as components failure rates.

## 5.2 Software Reliability Modeling

Two approaches for software reliability modeling are discussed in this section. These are the input domain modeling and the time-domain modeling approaches. In the development of high assurance systems, the behavior of redundant software modules for a given input stimulus is of great interest. This is the main focus of the input-domain reliability modeling approach. This approach evaluates the probability that the software will produce the correct answer. The definition of reliability, given in the previous section on the other hand, involves the behavior of hardware systems over time. The behavior of software over time is also of interest as software faults are identified and repaired. This motivates the time- domain reliability modeling approach.

### 5.2.1 Input-Domain Modeling

The input-domain approach [3, 42] is based on the execution of an input state as the index for the component or the software reliability. In the context of input domain modeling, reliability can be viewed as the number of successful executions over the total number of executions. The operational profile or the probability distribution of the input states is of special importance. The approach is based on identifying the events of interest and their estimated probabilities. A reliability model is constructed by combining these probabilities and obtaining an estimate of the software reliability. A simple example of reliability modeling for a recovery block (RB) system and N-version programming (NVP) system will be discussed. We consider a system composed of three redundant components.

In the recovery blocks structure, we let the reliabilities of the primary and two secondary redundant modules be $r_1$, $r_2$, and $r_3$, respectively. Let the reliability of the acceptance test be defined as $r_{at}$. The reliability of redundant modules are assumed to be estimated using the input domain approach based on unit testing. The reliability of the acceptance test is also estimated in a similar way. The system reliability estimate is obtained by estimating the reliabilities of the following events. The simplest event is when the primary produces the correct result and the acceptance test approves the output. The probability of this event is $r_1 r_{at}$. The second event is when the output of the first module is rejected and the output of the second module is accepted. This event can occur if the primary module produced a correct output which was falsely rejected by the acceptance test, or if the primary module produced an incorrect output what was correctly rejected by the acceptance test. The probability in this case is

$$r_1(1 - r_{at})r_2 r_{at} + r_{at}(1 - r_1)r_2 r_{at}.$$

The rest of the events are enumerated in a similar way and the overall system reliability is obtained. Assuming the modules have the same reliability estimate, the following estimate of the system reliability can be easily obtained,

$$R_{RB3} = r_{at} r(1 + r_{at} + r_{at}^2 + r + r^2 - 4r^2 r_{at} - 4r r_{at}^2 + 4r^2 r_{at}^2).$$

In the NVP system, the reliability is obtained by evaluating the probability that at least any two modules of the three redundant modules produce a correct output and the voter is correct. The system reliability obtained by enumerating the possible events and assuming the modules have the same reliability estimate $r$ and the voter has the reliability estimate $r_v$ is given by,

$$R_{NVP3} = r_v(3r^2 - 2 \times r^3).$$

The above equations show that under the assumptions of independent failures, the RB system is better than the majority voting three version system. This result was also obtained by Kanoun et al in [17] where it was also shown that the impact of common cause faults will likely be higher in the case of RB than in the case of NVP. This is because of the difficulty in developing very reliable acceptance test for RB systems, whereas, it may be far easier to develop very reliable voting software for NVP systems. Let us use the reliability estimate of the NVP system, $R_{NVP3}$, and compare it with the reliability of a non fault tolerant, single component (simplex) system. The reliability of a simplex system, denoted by $R_{simplex}$, is the same as the reliability of a single module in the NVP system, $r$. For the sake of argument, we will assume that reliability function $r$ follows exponential distribution (see Section 2.3.2), and that the voter in the NVP system cannot fail, i.e., $r_v = 1$[1]. The reliability of a system, according to Definition 17, is a probability that the system operates correctly during the predefined mission interval $[0..t]$. Hence,

$$R_{simplex}(t) = Prob(T > t) = \int_t^{\infty} \lambda\, e^{-\lambda x}\, dx \;=\; e^{-\lambda t}.$$

For the NVP system, reliability $R_{NVP3}$ is

$$R_{NVP3}(t) = 3e^{-2\lambda t} \;-\; 2e^{-3\lambda t}.$$

[1]The use of exponential distribution is common for hardware system, not for software systems. Therefore, the conclusions of this brief example are applicable for comparison between the hardware system employing triple modular redundancy technique and the equivalent single component system.

If we integrate $R_{simplex}(t)$ and $R_{NVP3}(t)$, we find that the mean time to failure (MTTF) measures of these two systems are given by

$$MTTF_{simplex} = \int_0^\infty e^{-\lambda t}\, dt \; = \frac{1}{\lambda},$$

and

$$MTTF_{NVP3} = \int_0^\infty (3e^{-2\lambda t} \; - \; 2e^{-3\lambda t})dt \; = \; \frac{5}{6\lambda}.$$

Thus, the MTTF of the $R_{NVP3}$ is lower than the MTTF of the simplex system. Based on these calculations, one might conclude that the fault tolerant NVP system is not as good as the non fault tolerant simplex system. This might and might not be true, depending on the specific application and the length of time the system is expected to function correctly. The instance of time $t_0$ in which the reliabilities $R_{NVP3}$ and $R_{simplex}$ are the same, can be calculated from equation

$$3e^{-2\lambda t_0} \; - \; 2e^{-3\lambda t_0} \; = \; e^{-\lambda t_0}.$$

Solving this, we get $t_0 = \frac{ln\ 2}{\lambda} = \frac{0.7}{\lambda}$. Hence, NVP configuration provides higher reliability only if the "mission time" of the system is less than $t_0$. From this simple calculation, a conclusion is that fault tolerant systems have their pitfalls, and the system design alternatives must be carefully studied before the choice is made.

In [32], a reliability model was developed to compare the techniques of software breeding (SB) and community error recovery (CER) in mltiversion software. Both SB and CER are techniques developed to increase the reliability of NVP. In CER, checkpoints are inserted at some predefined and homologous intermediate points in all program versions. The intermediate outputs at all program versions at a checkpoint are compared and set to the correct value to mask off any possible errors. The correct output is decided by majority voting. In SB, program versions are crossed by combining components of different versions to provide new program versions. The technique may detect and eliminate faults in program versions and produce more reliable versions. The model in [32] estimates the probability $P_e$ of the event that at least the minimum number of versions needed succeed on an input. For CER each version consists of $m$ modules with a checkpoint for each module. In this case the event of success must occur at each of the $(m-1)$ checkpoints. The reliability expression of CER is given by,

$$R_{cer} = R_v R_c^{m-1} P_e,$$

where $R_v$ and $R_c$ are the reliabilities of the voter and the checkpointing software, respectively. For the SB technique, for the breeding software to decide on correct module versions in major module decisions, it is necessary that more than half of the module versions succeed. In this case the expression for estimating $P_e$ will be evaluated accordingly. The reliability expression is given by,

$$R_{sb} = R_b P_e,$$

where $R_b$ is the reliability of the breeding software. The expression of $P_e$ given in [32] is derived based on assuming that failures in the corresponding module versions are independent and that the reliability of modules are identical. The results of the above study shows that CER improves the software reliability only when the number of modules in a version is relatively small. When the number of modules is large the reliability of the software is decreased because of checkpoint failures. In contrast, the SB technique increases the software reliability when the number of modules increases. However, it is clear that the reliability of the SB technique is bounded by the reliability of the breeding process which can be difficult to achieve and inefficient, with respect to realtime behavior

### 5.2.2 Time-Domain Modeling

This approach is concerned with the behavior of reliability over time as defined earlier for hardware systems. For example, during the process of testing and debugging, the reliability of software components is expected to grow, whereas during the operational phase without repair it may remain constant. Several groups of models have been proposed for this approach. The first group of models is based on the program failure [22] as the rate of the exponential distribution where the time between failures is modeled as an exponentially distributed random variable. The second

group of models is based on a reliability growth function, which predicts the improvement of reliability through the testing and debugging process. The independent variables of this growth function can be the time, number of test cases, or testing stages. The dependent variables are the reliability, the failure rate, and the cumulative number of errors detected [44]. The third group of models are Markov models where a non-stationary process can be used to unify many of the proposed models. The failure rate can be different for each fault. It can also be extended to model fault-tolerant techniques under different fault models. In the following paragraphs examples of Markov models for fault-tolerant software are discussed

In [10] a Markov model for NVP implemented on redundant hardware at the system level is proposed. The system modeled is designed to tolerate both hardware and software failures. The long-term behavior of the system is modeled by means of a Markov reward model. Each state of the model defines a configuration of successfully operational hardware and software components. A fault-tree model is developed for each state capturing the effects of software faults and transient hardware faults. The fault-tree model is constructed based on specific hardware and software models. The model supports independent software faults, related software faults, transient hardware faults, and permanent hardware faults. The software fault model is parameterized using experimental data associated with an NVP system implemented using the current design paradigm in which the prediction of software failures were very close to the empirical data. The hardware fault model is parameterized using typical failure rates associated with hardware faults and fault coverage parameters. The results of this study indicate that it is important to model both hardware and software faults in the reliability analysis of NVP systems. This is, for example, due to the effects of transient hardware faults on the behavior of the system and the voting process of NVP.

In [39], a Markov model of recovery blocks (RBs) was developed using Stochastic Petri Nets. The model captures correlated faults of three different types. The first type is the correlation between redundant software modules for a single input. The second type is the correlation between successive acceptance tests on correct module outputs and incorrect module outputs. The third type of correlated faults is due to clustering of difficult input in the input stream. The results of the model shows that clustered failures have a negative impact on the reliability of RBs. It is also shown that the greatest benefits in reliability improvement is when the number of modules in RBs is increased up to 3; beyond 3 the improvement is quite small. Finally, the incorporation of limited correlation between redundant modules has a significant impact on reliability as expected, in terms of mean time between failures.

## 5.3 Reliability Measurement

Measurement based analysis of reliability in the operational phase has made significant progress in the last 15 years [11]. Results from experimental analysis have been used to validate analytical models. They have also been used to reveal important practical issues that must be addressed and have been purposely or accidentally overlooked by analytical models. Failure data from the operational phase has been used to identify software design issues for improving software reliability. In addition, efficient on-line recovery and off line diagnosis techniques are being developed based on data collected from the operational phase.

Measurement techniques at the operational phase are based on on-line logging of errors during the system operation or manual reporting. Operating systems are run-time environments, usually equipped with software for on-line error logging. Manual reports are usually generated by users, problem analysts, and developers. Measurement techniques in the operational phase or in the development phase are confronted with traditional theoretical and practical problems. A measurement strategy should be non-intrusive. A large amount of data must be collected for statistical purposes. In fault-tolerant systems failures are infrequent and to obtain meaningful data, we must deploy the measurement process for a long time period. The design of representative workloads or benchmarks needed to expose the measured system to a wide range of conditions is also a difficult task.

Relatively little research has been performed in the area of fault tolerance benchmarks. A benchmark for measuring fault tolerant systems at the development phase was proposed in [41]. The benchmark consists of a synthetic program that generates CPU, memory, and I/O activity based upon user specifications. It also injects CPU, memory, and I/O faults according to a specified injection strategy. The benchmark supports two types of metrics, the number of catastrophic incidents and performance degradation. The benchmark has been used to measure two TMR-based prototype systems. The results demonstrate the utility of the benchmark in comparing the system-level fault tolerance of these systems and in providing insight into their design. In particular, the benchmark shows that the first Prototype

suffers fewer catastrophic incidents than the second Prototype under the same workload conditions and fault injection method. However, the second Prototype also suffers more performance degradation in the presence of faults, which might be an important concern for time-critical applications.

Several approaches have been proposed for measurement techniques based on fault injection. Software-implemented fault injection (SWIFI) is a technique proposed for measuring fault tolerant hardware. This approach uses software to emulate the effect of lower level hardware faults by altering the contents of memory or registers. Several fault injection tools have been developed based on this approach. FTAPE (Fault Tolerance And Performance Evaluator) is a tool developed based on SWIFI to measure fault-tolerant systems [41]. The tool includes a system-wide fault injector, a workload generator, and a workload activity measurement tool. The workload creates high stress conditions on the system under test. Using stress-based injection, the fault injector utilizes the knowledge of the workload activity to ensure a high level of fault propagation. The error ratio, fault ratio, performance degradation, and number of system crashes are considered as measures of fault tolerance.

DEPEND is a simulation-based environment for the evaluation of software/hardware designs from functional and reliability viewpoints [29]. The environment provides a graphical modeling facility to allow interactive model construction. The environment supports functional or structural descriptions of the designs. The Task and Environment Manager provides an easy-access to a tool-kit to allow users to create, modify, and simulate models. The following techniques are supported:

- Functional-level specification techniques
- High-level synthesis-for-test techniques
- Multidimensional fault sensitivity analysis: This includes chip-level analysis using importance sampling and concurrent simulation, as well as system-level analysis to model fault impact at the software/ system level.
- System-level reliability analysis: This includes time acceleration, hybrid fault-sensitivity simulation, and software characterization under fault conditions.

DEPEND employs a hierarchical modeling and simulation approach. This approach is intended to allow the evaluation of designs starting from the lowest level, that is, the device-level physical construct, to the chip-level functional behavior, and up to the system-level dependability. At the core of DEPEND are simulation engines supported by a fault injector, a set of fault dictionaries, and component libraries. The component libraries contain model building blocks with detailed functional descriptions and characteristics; the fault dictionaries embody possible fault effects of given fault types, devices, and circuits; and the fault injector provides mechanisms to inject faults. With the help of fault dictionaries available at different abstraction levels, faults can be injected at the lower level, such as the device or gate level, and effects are evaluated at the chip or system levels.

# 6 Conclusion

## 6.1 Summary

As computer systems grow increasingly complex, they also become increasingly likely to have faults, stemming from their requirements specification, their design, their implementation, or their operating environment. On the other hand, as computer systems take on more and more crucial functions, their failure-free operation grows increasingly indispensable. Fault tolerance includes the set of provisions that make it possible for a system to operate without failure despite the presence and occasional sensitization of faults. In this paper we have presented some definitions of system fault tolerance, and discussed some issues that arise in hardware fault tolerance and software fault tolerance. We have also analyzed the impact of fault tolerance requirements on system design, and the impact of fault tolerance provisions on system reliability.

## 6.2 Lessons Learned

One of the main lessons of our analysis is that hardware fault tolerance and software fault tolerance are governed by essentially the same fundamental concepts. We have found it easier to discuss these concepts in the context of

software, because software affords us a sharp distinction between control and data (by means of executable statements vs state variable declarations), whereas hardware blurs this distinction (the same register may be interpreted as data or as control, at different stages of the same execution process). From the viewpoint of fault tolerance, the distinction between control and data is crucial, because faults are features of the control whereas errors are features of the data.

The deployment of these fault tolerance concepts yields a wide variety of techniques, due to the wide range of hypotheses and conditions under which these concepts are deployed. Another lesson to draw from our discussions is the use of redundancy, under a variety of forms, to achieve fault tolerance: the design of a fault tolerant system involves a set of tradeoffs between redundancy requirements (imposed by the need for fault tolerance) and requirements of economy (economy of the process, and the product). Other specific lessons include:

- Fault tolerant techniques are designed to enhance the system reliability but they complicate reliability modeling and analysis. This is due to the fact that reliability models of fault tolerant systems need to assess the effects of different types of redundancies and different types of correlated failure.
- New techniques are needed to effectively model reliability of fault-tolerant systems in which the fault models and recovery processes of hardware and software components are adequately addressed. Techniques based on combining stochastic Petri Nets with Fault trees are among the promising approaches suggested.
- Measurements and benchmarking techniques fault injection of prototypes and operational systems has made significant progress in comparing and evaluating the system level fault tolerance.
- System level reliability analysis environments based on hierarchical modeling and simulations are emerging as powerful tools for assessing the reliability of fault-tolerant designs. These environments include functional level specification techniques, fault injectors, fault dictionaries, and components libraries containing model building blocks with functional description and characteristics.
- Like most everything in software engineering, program fault tolerance is no panacea. While it does, in principle, offer some protection against the impact of faults, fault tolerance comes at a cost, in terms of increased complexity, increased time and space overhead, and ultimately, a reliability penalty. So much so that it is possible to find cases where fault tolerance provisions actually degrade reliability rather than enhance it (see section 5.2.
- Redundancy management has costs associated with it: operating cost (performance), development cost, as well as additional complexity. Spatial redundancy in hardware tends to be cheaper in terms of development costs, because (1) hardware faults are often expected to be independent and due to wear, and (2) it is less expensive to manufacture identical hardware components than to produce diverse software designs.

# References


[1] L. Alkalai and A. T. Tai. Long-life deep-space applications. *Computer*, 31(4):37, April 1998.

[2] A. Avizienis. The n-version approach to fault tolerant software. *IEEE Trans. on Software Engineering*, 11(12), December 1985.

[3] F.B. Bastani and C. V. Ramamoorthy. Input-domain-based models for estimating the correctness of process control programs. In *In Reliability Theory*. North-Holland, Amsterdam, 1986.

[4] P. G. Bishop. The variation of software survival time for different operational input profiles. In *Proc. IEEE Int'l Symp. on Fault-Tolerant Computing FTCS-23*, pages 98–107, Toulouse, France, June 1993.

[5] P. G. Bishop and F. D. Pullen. Pods revisited - a study of software failure behavior. In *Proc. IEEE Int'l Symp. on Fault-Tolerant Computing FTCS-18*, Tokyo, Japan, June 1988.

[6] S. S. Brilliant, J. C. Knight, and N. G. Leveson. The consistent comparison problem in n-version software. *IEEE Transactions on Software Engineering*, 15(11):1481–1485, November 1989.

[7] Mike Carlton. Pentium divide bug faq. Technical report, Intel Corp., June 1995.

[8] Payload Integration Company. Space systems performance, endurance and survivability. Technical Report PL-TR-97-C-1141, Report submitted to U.S. Air Force, Phillips Laboratory, = Space Missile Technology Directorate, 1997.

[9] Intel Corporation. Floating point division with optional checking to ensure full result precision, january 1995.

[10] J.B. Dugan and M.R. Lyu. System reliability analysis of n-version programming applications. In *International Symp.on Software Reliability Engineering*, Denver, Colorado, November 1993.

[11] M.R. Lyu (ed). *The Handbook of Software Reliabilty Engineering*. IEEE Computer Society Press, 1996.

[12] G. B. Finelli. Results of software error-data experiments. In *AIAA/AHS/ASEE Aircraft Design, Systems and Operations Conference, AIAA'88*, Atlanta, GA, 1988.

[13] H. Hecht and P. Crane. Rare conditions and their effect on software failures. In *Proc. Ann. Reliability and Maintainability Symp.*, pages 334–337, Anaheim, CA, January 1994.

[14] H. Hecht, M. Hecht, and D. Wallace. Toward more effective testing for high assurance systems. In *Proceedings of the 2nd IEEE High Assurance Systems Engineering Workshop*, Washington, D.C., August 1997.

[15] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, modules and applications. In *The 25th International Symposium on Fault-Tolerant Computing (FTCS 25)*, pages 381–390, Pasadena, CA, June 1995.

[16] B. W. Johnson. *Design and Analysis of Fault Tolerant Systems*. Addison Wesley, 1989.

[17] K. Kanoun, M. Kaaniche, C.Beounes, and J.C. Laprie. Reliability growth in fault-tolerant software. *IEEE Transactions on Reliability*, 42(2):205–219, June 1993.

[18] J. Laplace and M. Brun. Critical software for nuclear reactors: 11 years of field experience analysis. In *The 9th International Symposium on Software Reliability Engineering, (ISSRE'98)*, pages 364–368, Paderborn, Germany, November 1998.

[19] P. R. Lorczak, A. K. Koglayan, and D. E. Eckhardt. A theoretical investigation of generalized voters for redundant systems. In *The 19th International Symposium on Fault-Tolerant Computing (FTCS 19)*, pages 444–451, 1989.

[20] T.E. Mangir and A. Avizienis. Fault tolerant design for vlsi: Effect of interconnect requirements on on yield improvement of vls designs. *IEEE Transactions on Computers*, C-31(7):609–616, 1982.

[21] A. Mili. *An Introduction to Program Fault Tolerance: A Structured Programming Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[22] P.B. Moranda. An error detection model for application during software development. *IEEE Transactions on Reliability*, R-28(5):325–329, 1979.

[23] J. C. Munson and T. M. Khoshgoftaar. Software metrics for reliability assessment. In *In The Handbook of Software Reliability Engineering*. IEEE Computer Society Press, 1996.

[24] J. D. Musa. *Software Reliability Engineering*. McGraw-Hill, 1998.

[25] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Applications*. McGraw-Hill (professional edition), 1990.

[26] D. K. Pradhan. *Fault Tolerant Computing: Theory and Practice*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[27] B. Randall. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2), 1975.

[28] ESA Release. Ariane 501 —presentation of the inquiry board report. Technical Report 33-96, European Space Agency, Paris, France, July 1996. Also available at `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/press33.html`.

[29] G. Ries, Z. Kalbarczyk, T. Kraljevic, M.C. Hsueh, and R.K. Iyer. Depend: A simulation enviroment for system dependability modelling and evaluation. In *IPDS'96: IEEE International Performance and Dependability Symposium*, september 1996.

[30] P. B. Selding. Faulty software caused ariane 5 failure. *Space News*, 7(25), June 1996.

[31] H.P. Sharangpani and M.L. Barton. Statistical analysis of floating point flaw in the pentium processor. white paper, Intel Corporation, november 1994.

[32] K. Shima, K. Matsumoto, and K. Torii. A mathematical comparison of software breeding and community error recovery in mutliversion software. In *International Symp.on Software Reliability Engineering*, Denver, Colorado, November 1993.

[33] Daniel P. Siewiorek and Robert S. Swarz. *Reliable computer systems : design and evaluation*. A K Peters, Natick, Mass., 3rd edition, 1998. 98020237 Daniel P. Siewiorek, Robert S. Swarz. Rev. ed. of: The theory and practice of reliable system design. Bedford, MA : Digital Press, c1982. Includes bibliographical references (p. 845-884) and index.

[34] D.R. Skuce and A. Mili. Behavioral specifications in object oriented programming. *Journal of Object Oriented Programming*, pages 41–49, January 1995.

[35] DoD Staff. Military standardization handbook: Reliability prediction of electronic equipment. Technical Report MIL-HDBK-217C, The US Department of Defense, 1979.

[36] W. Stallings. *Computer Ogrnzation and Architecture: Principles of Structure and Function*. McMillan, New York, NY, 1987.

[37] A. T. Tai and L. Alkalai. On board maintenance for long-life systems. In *1998 IEEE Workshop on Application SPecific Software Engineering and Technology*, pages 69–74, Dallas, TX, March 1998.

[38] D. Tang and R.K. Iyer. Analysis of VAX/VMS error logs in multicomputer environments —a case study of software dependability. In *Third International Symposium on Software Reliability Engineering*, pages 216–226, Research Triangle Park, NC, 1992.

[39] L.A. Tomek, J.K. Mupalla, and K.S. Trivedi. Modelling correlation in software recovery blocks. *IEEE Transactions on Software Engineering*, SE-19(11), november 1993.

[40] K. S. Trevedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[41] Timothy K. Tsai, Ravishankar K. Iyer, and Doug Jewett. An approach towards benchmarking of fault tolerant commercial systems. In *FTCS'96: Fault Tolerant Computing Symposium*, Sendai, Japan, 1996.

[42] S. N. Weiss and E.J. Weyuker. An extended domain-based model of software reliability. *IEEE Transactions on Software Engineering*, SE-14(10):1512–1524, october 1988.

[43] V. L. Winter, J. M. Covan, and L. J. Dalton. Passive safety in high-consequence systems. *Computer*, 31(4), April 1998.

[44] S. Yamada and S. Osaki. Software reliability growth modeling: Models and applications. *IEEE Transactions on Software Engineering*, SE-11(12):1431–1437, december 1985.