Evaluation & Comparison of Fault-Tolerant Software Techniques

John Hudak
Carnegie Mellon University, Pittsburgh
Byung-Hoon Suh
Carnegie Mellon University, Pittsburgh
Dan Siewiorek
Carnegie Mellon University, Pittsburgh
Zary Segall
Carnegie Mellon University, Pittsburgh

Key Words — **Fault-tolerant software, *n*-version programming, Recovery block, Concurrent error-detection, Algorithm-based fault tolerance, Markov model, Fault-injection testing**

Reader Aids —
Purpose: Compare various techniques
Special math needed for derivations: Markov modeling
Special math needed to use results: Same
Results useful to: Software fault-tolerance practitioners & researchers

Summary & Conclusions — **Various fault-tolerant software techniques have been proposed in order to meet the reliability requirements of critical systems. This paper evaluates 4 implementations of fault-tolerant software techniques with respect to hardware and design faults. Project participants were divided into 4 groups, each of which developed fault-tolerant software based on a common specification. Each group applied one of the following techniques: *n*-version programming, recovery block, concurrent error-detection, and algorithm-based fault tolerance. Independent testing and modeling groups within the project then thoroughly analyzed the fault-tolerant software. Using fault-injection tools, the testing group subjected the fault-tolerant software to simulated design and hardware faults. Simulated design-faults included control flow, array boundary, computational, and post/pre increment/decrement software mutations. Simulated hardware-faults included code and data corruption. Data collected from the fault-injection experiment were then mapped into a discrete-time Markov model developed by the modeling group. Based on this model, the effectiveness of each implementation of the fault-tolerant software technique with respect to availability, correctness, and time to failure given an error, is contrasted with measured data. Finally, the model is analyzed with respect to additional figures of merit identified during the modeling process, and the techniques are ranked using an application taxonomy.**

1. INTRODUCTION

Computers are being used in more critical-applications where computer failure is costly in terms of time, material, or even lives. A study of Tandem Computers systems [4] shows the time rate of change of the source of system outages. Software was the source of 33% of the outages in 1985 and grew to 62% of the outages in 1990. The Tandem study motivates our study of fault-tolerant software techniques. Tandem systems deal with on-line transaction processing and access to large databases. Due to the complex behavior of full commercial operating systems and databases, it is not possible to examine fully a system of that size. Conversely, an application that might simulate the behavior of a small portion of a larger system can be thoroughly studied.

This paper describes the results of an experiment at Carnegie Mellon University that compared several fault-tolerant software techniques. It has been argued that current validation techniques are insufficient to meet the reliability goals required by today's critical systems. The argument further claims that some form of redundancy is needed to allow systems to overcome faults that might be encountered during operation. The redundant approach to fault tolerance is examined in these experiments. Different implementations of fault-tolerant software were independently developed using a common specification. These implementations were then injected with faults that simulated design (software mutations) and hardware (memory corruption) faults. Program behavior was then mapped to a unified model to compare the effectiveness of the fault-tolerance techniques.

All participants in the various groups were graduate students.

Acronyms

AFT	algorithmic fault-tolerance
base/imp	baseline implementation(s)
CDMM	collapsed discrete-time Markov model
CED	concurrent error-detection
FTS	fault-tolerant software(s)
FTS/imp	FTS implementation(s)
FUV	final unlocking vector
GDMM	generalized discrete Markov model
LIC	launch interceptor conditions
LIP	Launch Interceptor Program
MTTFGE	mean time to failure, given error exists
N-V	*n*-version
PUM	preliminary unlocking matrix
RB	recovery block
SE	software engineering.

Nomenclature

fault. Direct cause that, if implemented, leads to an error.
error. An anomalous program state that is the result of an execution of a fault and can lead to a failure.
failure. Delivered results are different from the anticipated results of the specification.
FTS/imp. Program that implemented 1 of the 4 FTS techniques.
FTS development group. One of 4 groups of graduate students that developed FTS/imp in phase IIa.
FTS technique. A theoretical approach that was applied in FTS/imp during phase IIa.

0018-9529/93/$3.00

Phase I. A target application specification was used to develop non-FTS (normal) implementations. A total of 6 implementations were written: 5 implementations by groups of two graduate students, and 1 SE (software engineering) implementation by a group of seven graduate students. The SE implementation was written applying rigorous SE guidelines. The other groups used *ad hoc* development methods (no formal methodology). All software implementations went through a thorough testing/validation process by a separate testing group. The SE implementation was used in phases IIb & IIc as a non-FTS implementation for comparison purposes (the baseline).

Phase II. Phase II was divided into 3 subphases: IIa) FTS/imp, IIb) testing, and IIc) modeling. Phase IIa began by reorganizing the participants from phase I into 6 new groups: 4 FTS development groups for FTS/imp, 1 testing group, and 1 modeling group. The 4 FTS development groups each wrote a FTS/imp of the application specification using as a basis the verified implementations of phase I. Each FTS development group was assigned a different FTS technique to implement:

- *n*-version programming [2],
- recovery block [11],
- concurrent error-detection [9],
- algorithmic fault-tolerance [5,19].

The 4 FTS development groups were advised that: 1) the FTS/imp they developed would be subjected to simulated design or hardware faults, one at a time, and 2) neither performance nor overhead would be considered. When each FTS/imp was complete, it was given a preliminary acceptance test. If it passed the preliminary test, the FTS/imp was submitted to the testing group for a very thorough final test. During the final-test of phase IIb, information concerning code execution (error detection, recovery, abort) was collected and given to the modeling group. In phase IIc the modeling group analyzed these data and compared the FTS techniques. As this was the first time that these students had developed a FTS/imp, the results should represent typical first attempts.

The following sections detail the results of this experiment. Section 2 presents the target application. Section 3 describes the design and implementation of the FTS techniques by each FTS development group. Section 4 presents test methodologies used to final-test the FTS/imp while extracting data for modeling. Section 5 describes the general purpose state model. Section 6 presents data acquisition, reduction , and analysis techniques used to complete the model. Sections 7 - 10 present results & observations for the 4 FTS/imp.

2. TARGET APPLICATION

Before the project started, it was determined that our specification should meet the following guidelines:

- previously used and thoroughly debugged,
- appropriate size[1],
- clearly worded,
- non-trivial.

[1]The program had to be written in the first of 2 phases of a 3-month course, by groups as small as 2 people.

Three sets of experimenters were asked to provide their most current & thorough specifications. The resultant specifications were in various stages of development and focused on diverse applications. The specification chosen was the Launch Interceptor Program (LIP) [7] since, at the time the course was offered, it best fit our guidelines.

The LIP is part of a hypothetical antiballistic missile system that determines if an interceptor should be launched at incoming objects being tracked by radar. There are several data structures defined in the specifications, illustrated in figure 2.1, and mentioned frequently in this paper. The decision to launch is based on 15 launch interceptor conditions (LICs). Each LIC computes a *true* or *false* value based on the input points from the radar. The resulting 15 Boolean LIC values form the conditions met vector (CMV) containing 15 entries. A symmetric 15 x 15 Logical Connector Matrix (LCM) is given as input and determines how CMV elements are considered jointly through *and, or,* or *not-used* operators. The results of all of these operations are stored in the off-diagonal locations of the 15×15 preliminary unlocking matrix (PUM). The diagonal elements of the PUM are given as input to the LIP and specify how off-diagonal elements are combined to form the 15-element final unlocking vector (FUV). A decision to launch is made only if all elements of the FUV are *true*. The variable LAUNCH is set equal to this decision. Noting the short-comings of a single binary output, the specification was modified to include all CMV values as output as well.

3. FTS TECHNIQUES

This section briefly describes each of the 4 FTS techniques. The descriptions are brief because development of new FTS techniques was not a project goal.

3.1 *Algorithmic Fault-tolerance*

AFT (algorithmic fault-tolerance) describes techniques in which the software fault tolerance is tailored to the algorithm to be performed [5]. In [5] each matrix is augmented with a row or column checksum (row or column checksum matrix) such that matrix operations result in a matrix that contains a core that is augmented by both row & column checksums. In [19,20] linked lists with redundant links and special identification fields are used within list items. For the LIP, a 2-level AFT approach was chosen that separated the goals of: 1) data storage integrity, and 2) operation correctness. Storage of LIP global variables was protected by checksums on the matrices; correct operation was checked with *trial calls*. Trial calls are defined at the functional level, and involve execution of a function using data for which correct results are known and stored as anticipated results. Before execution of each function, a trial call is made and the results are compared to the anticipated results. If they disagree, an error is signaled and the program is aborted.

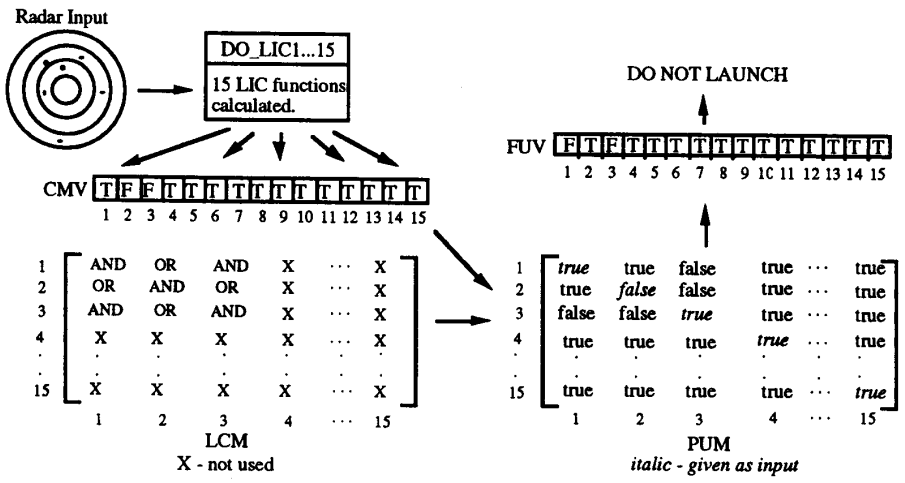

Figure 2.1. LIP Matrices & Vectors

3.2 *Concurrent Error-Detection*

CED (concurrent error-detection), in a broad characterization, describes techniques in which errors are detected implicitly (by a watchdog) during program execution. In software, a watchdog could be a separate process that runs concurrently with the main process [9]. During execution of the software, the watchdog monitors the process or processor and collects relevant data. An error is detected when the watchdog detects a discrepancy between anticipated behavior and actual execution. Assertions were used as the primary error detection mechanism in this experiment. Assertions are logical statements at various points of the program. Assertions evaluate a condition that the program designer believes to be always *true* when executed. An error is detected when an assertion executed by the watchdog evaluates to *false*. For the LIP, one assertion was inserted after every major block of computation as well as within and after every loop.

3.3 *N-Version Programming*

The N-V programming approach is characterized by different programming groups generating $n \geq 2$ software modules (versions) from the same set of requirement specifications [2,7] in order to minimize software design & implementation errors. Each version is executed in parallel and the output from each version is directed to a voter module. The voter module rectifies any disagreement of the results. The N-V FTS development group used the 6 versions of the LIP that were implemented in project phase I. A 3-level N-V voting scheme was developed wherein the results of 5 of the versions were fed to 3 voters whose results were then input to the controller for a final decision. If a vote by 1 voter is ambiguous, the SE implementation result is used twice and the final vote is recalculated. To make the communication mechanism more fault tolerant, a "distance-8 linear Hamming code over GF(2)" was implemented which is equivalent to extending the value of a data bit into an entire byte [12].

3.4 *Recovery Block*

The RB (recovery block) paradigm [11] essentially adapted a hardware redundancy technique (standby sparing) for use in software. Standby sparing allows different implementations of a system to be available for use; however only 1 implementation is used at any time, and the other implementations are used only when the main system is found to be operating incorrectly. In RB, the application program is divided into subprograms (blocks). Each block has several implementations available for execution. The implementation invoked first is the *primary*, and the other implementations are *alternatives* to the *primary*. Results from the execution of the *primary* are supplied to an *acceptance test* which verifies the accuracy of the results. Upon passing the *acceptance test* the next subprogram in the *primary* is executed. If the results do not pass the *acceptance test* the state of the process is restored to that just prior to the execution of the *primary* at the subprogram level, and an *alternative* is executed. *Alternatives* continue to be executed until a result passes the *acceptance test*, or, if all the *alternatives* fail the *acceptance test*, then a failure is reported. For the LIP, 1 *alternative* was deemed appropriate since only single faults were to be injected into thoroughly verified implementations.

4. TESTING OF THE FTS IMPLEMENTATION

There have been no extensive publications on the testing requirements of FTS/imp. Our approach was a cooperative effort between the modeling and testing groups. The two groups together defined the set of faults against which fault tolerance would be measured. The modeling group defined internal states, such as detection-of & recovery-from errors, that the testing group should measure. The result represents a pioneering effort in the use of a mutation analysis system (FAUST) [18] and a fault injection testing system (FIAT) [14] to validate FTS/imp.

4.1 *Testing of FTS Implementation*

Each FTS/imp was subject to fault injections and the results compared with a thoroughly tested *golden standard* (the SE implementation of phase I). A testing harness was built to:

- run the experiments,
- detect transitions between internal states[2] of the implementations,
- detect & kill programs that went into an infinite loop.

Portions of two existing testing systems were used as part of the testing harness: 1) FAUST [18] was used to inject design faults into the implementation by simulating small syntactic changes through programming language statement mutations; 2) FIAT [14] was used to simulate the effects of hardware faults by changing the values of memory locations. Table 4.1 lists the fault classes.

TABLE 4.1
List of Injected Fault Classes.

Design Faults (PL Statement Mutations)	Hardware Faults (Memory Fault Injection)
Control flow	Code
Array Boundaries	
Mathematical	Global data of FTS/imp
Pre/post incr/decrement	

PL = Programming Language

To achieve fairness in testing, the same number of faults were inserted into each FTS/imp. The FAUST system was used to measure fault-tolerance coverage against design-faults (mutations). Since the number of mutations depends on the number of mutable operators in the implementations, the number of mutations was different for each FTS/imp as depicted in table 4.2. For each FTS/imp, 100 mutations were selected at random and run with 207 test vectors. In FIAT testing, 80 data faults (in global variables) and 20 code faults (in procedure headings) were inserted into each FTS/imp. Of the 80 data faults, 70 were for global variables used in the LIP and 10 were for the global variables used in the error detection and recovery code. The 20 code faults were inserted into randomly chosen functions in the FTS/imp. Either a bit or byte of a memory location was corrupted in order to simulate both chip and bus faults.

TABLE 4.2
Mutations Identified For Each FTS Implementation

FTS Implementation	Number of mutations
Baseline	412
Algorithmic Fault-Tolerance	2074
Concurrent Error-Detection	2760
Recovery Block	3084
N-Version	6674

[2]Such as error detection and recovery as defined in the model of figure 5.1.

4.2 *Test Execution Results*

For all FTS/imp, testing was initially performed using FAUST and then FIAT. Since the applied FTS techniques result in different behavior for the FTS/imp, the results are presented separately in the following paragraphs. Figures 4.1 & 4.2 show the fault-execution percentages for FAUST (design faults) and FIAT (hardware faults) experiments. A fault is activated if a given test set causes its execution; when a fault is activated it results in an error. A error is detected if an activated fault causes an error detection sensor to be triggered.

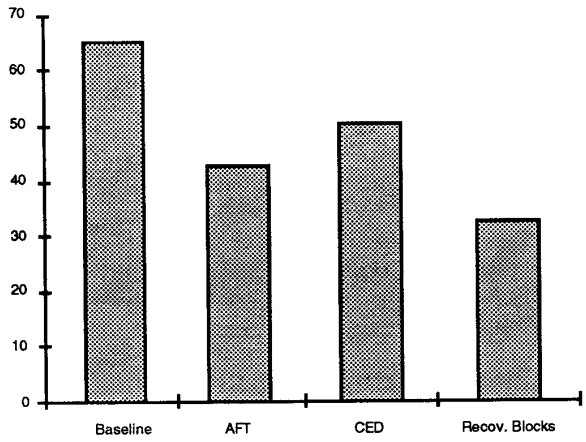

Figure 4.1. Percentage of Faults Executed & Detected in FAUST Experiments

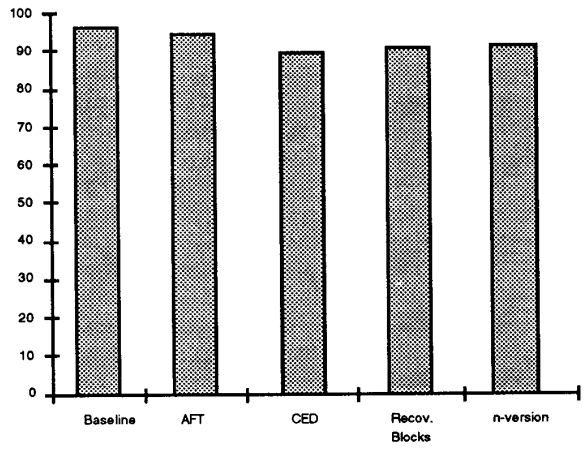

Figure 4.2. Percentage of Faults Executed & Detected in FIAT Experiments

Design-fault tests were not conducted on the N-V FTS/imp due to time constraints. Considering the testing process, we can deduce that at most 2% of the FAUST-faults injected would

produce incorrect results[3]. Due to limitations on the number of active processes imposed by the FIAT system, the simulated hardware-fault testing for the N-V FTS/imp was carried out with 5 processes instead of the original 10 (3 versions, 1 voter, 1 controller). The 5-process system should be appreciably less tolerant to faults than the original 10-process implementation.

The executed fault-injection figures are higher for FIAT than FAUST because most FIAT-faults target *global* variables whereas FAUST-faults target code. Due to the nature of the LIP (15 different LICs that evaluate to either *true* or *false*), 100% coverage is not possible with any single test case. Test cases are tailored to target a few of the LICs at a time in which case others might be missed. Since the faults injected to each FTS/imp are different, no attempt was made to tailor test-cases to individual fault injections. Instead, a large test-set was developed to be applied to all FTS/imp and their associated fault-injections.

5. MODELING OF THE FTS IMPLEMENTATIONS

The previous sections have presented 4 FTS/imp and the testing methodology applied to each. For each injected fault, 100s of test vectors were executed and the output recorded for modeling.

5.1 *Modeling Methodology*

The modeling effort was directed towards:

- establishing a common model for all of the FTS/imp
- subsequently evaluating & comparing the FTS techniques.

The remaining portion of this paper discusses the methodology, tradeoffs, analysis, and results in evaluating the 4 FTS/imp.

The concepts of state and state-transitions are the basis for Markov chains [6,8,21]. From a survey of modeling techniques designed for hardware systems it became apparent that software programs have no "tangible" components analogous to hardware components (*eg*, transistors, ICs). Methods to measure the reliability of such hardware components have been well defined. Software reliability, on the other hand is not as well defined.

Subsequent investigations suggested that the "state" of a software program could be modeled as a virtual component [1,3,21]. Although the concept of state was straight-forward, a concise & accurate definition of state was difficult to establish. It was highly desirable to establish one common state-model that would reflect all the FTS/imp, thereby providing a common reference from which all evaluations could be performed. For an initial modeling effort, a few states were desirable. These states would be used to evaluate the chosen criteria. To calibrate & validate the model, data would be collected from the FTS/imp fault-injection experiments. Therefore, the chosen states must be *measurable* with the tools available.

[3]Due to redundancy in voters & versions, only mutations to the controller can cause incorrect output. These mutations account for less than 2% of the total number of faults.

A preliminary Markov model was developed that would apply to all FTS/imp. The states of the preliminary model can be classified as either *operational* or *completion*. The operational states are:

- Good. The FTS/imp is executing with no faults activated;
- Error. The FTS/imp is in a state of internal error due to activation of a fault;
- Detected. An error has been detected, and the FTS/imp is attempting a recovery;
- Recovered. The FTS/imp has taken corrective action to mitigate the error.

The completion states of a FTS/imp are:

- Passed. The FTS/imp has completed and returned a correct result;
- Failed. The FTS/imp has completed and returned an incorrect answer it believes to be correct;
- Abort. The FTS/imp is unable to recover from a detected error and has halted itself; it is assumed the operating system kernel is still functioning;
- Crash. The result of an error is that a catastrophic system-failure has occurred.

In the initial design of the model, the FTS/imp returned to the Good state after an error had been Detected. If the error occurred within a loop of the FTS/imp, however, the FTS/imp would repeatedly make the Good → Error transition, overbiasing that arc and skewing the actual probability of encountering an error. By adding a Recovered state, the probability of encountering the same error repeatedly was separated from the probability of initially encountering the error. Figure 5.1 shows the final GDMM.

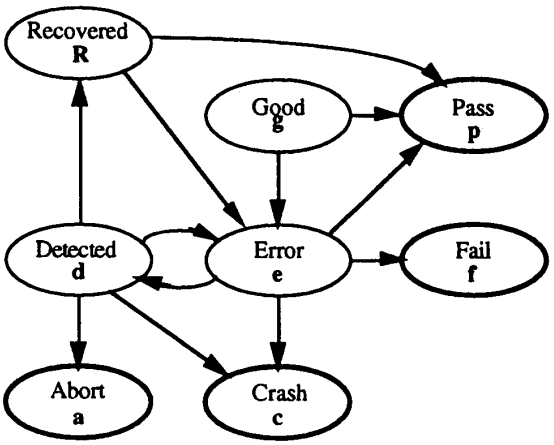

Figure 5.1. Generalized Discrete Markov Model (GDMM)

The arc traversals can be interpreted as follows. When the FTS/imp begins execution, it is assumed to be in the Good state. Suppose that an injected fault is activated during FTS/imp execution. This action forces the transition to the Error state. From

this state, several possibilities arise. The error might be catastrophic, causing the system to crash, hence the presence of the Error → Crash arc. The error might not be detected, causing an incorrect result, arc Error → Fail. The error might be trivial or be masked by later actions of the FTS/imp, allowing the FTS/imp to return a correct answer, arc Error → Passed (a lucky outcome). Finally, the FTS/imp might detect the error, placing the FTS/imp in the Detected state.

Once in the Detected state, the FTS/imp attempts to recover. For some FTS techniques, such as CED, there is no ability to recover and the FTS/imp simply aborts gracefully (Abort). Otherwise a recovery mechanism is invoked. If the recovery is successful, the FTS/imp moves to the Recovered state, where it might complete correctly, or encounter another error. If the recovery mechanism fails to correct the error, the FTS/imp returns to the Error state.

Several figures of merit for the FTS/imp can be obtained directly from the GDMM arc probabilities:

- Coverage. Pr{detecting an error | a fault is active}, Error → Detect;
- Recovery. Pr{recovering successfully after error detection}, Detect → Recovered;
- Aborting. Pr{aborting | successful recovery is not possible}.

Assumptions (inherent in our Markov model)

1. The FTS/imp was completely fault-free prior to the injected fault. Extensive tests were performed on the base/imp (non-FTS implementations of phase I). Even though an appreciable amount of code was added to the base/imp to create the FTS/imp and connect them into the test harnesses, subsequent analysis of the program state transitions revealed that extremely few impossible transactions were discovered. Therefore the fault-free assumption was still considered appropriate.

2. Only a single fault is injected. This was necessary to keep the model simple. The testing with FAUST & FIAT specified that only one fault was to be injected for each run.

3. The model is a discrete-time model. All events happen at constant intervals. Some evaluation criteria, such as mean time-to-failure are time dependent. In order to include the time factor, it is assumed that each complete execution of the FTS/imp takes one unit of time. In reality, this is not true; however, for modeling purposes the average execution time can be used as the unit-time step. ◄

5.2 *The Collapsed Discrete-Time Markov Model (CDMM)*

The CDMM was devised to view easily the probability of FTS/imp execution's terminating in 1 of the 4 failure states. The internal states of the FTS/imp were collapsed to the good state as shown in figure 5.2. The probabilities of each arc transition is indicated by P_n.

The primary advantage of this model is that only the completion state of the FTS/imp need be measured to determine the arc probabilities. This model successfully integrates the possibility of recovery loops in the GDMM into a single transition, making the probability of reaching each completion state readily apparent. The 4 transition probabilities depicted in figure 5.2 are:

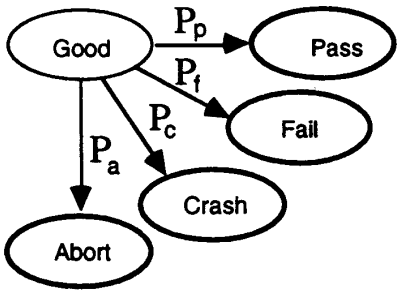

Figure 5.2. The Collapsed Discrete-Time Markov Model

- P_p Pr{transition to the Pass state}
- P_f Pr{transition to the Fail state}
- P_c Pr{transition to the Crash state}
- P_a Pr{transition to the Abort state}.

The CDMM can be created directly from the GDMM by computing the n-step transition matrix for the GDMM and letting $n \rightarrow \infty$. The n-step transition matrix is found by raising the 1-step transition matrix to the n^{th} power.

The figures of merit derived from the CDMM are:

- Correctness. Pr{a wrong answer is not reported}; $(1-P_f)$;
- Availability. Pr{reporting the correct answer or aborting}; this definition assumes that the program is still providing its service if it is able to consciously and gracefully exit or issue a *try again* message; (P_p+P_a).

5.3 *The Continuous-Time Markov Model*

The arc transition probabilities in section 5.2 can be related to time, resulting in transition rates. From assumption #3 (section 5.1), all executions take one unit of time (including multiple passes in the Error Detected → Recovery cycle)[4]. The failure rate is then the probability of complete failure in one unit time. From the CDMM, this is: $P_f + P_a + P_c$, which was used as the basis for the 2-state reliability model in figure 5.3.

The figure of merit for this model is MTTFGE (mean time to failure, given error exists). For a 2-state model MTTFGE is the reciprocal of the failure rate [16]:

$$\text{MTTFGE} = 1/(\text{failure rate}) = 1/(\lambda_f + \lambda_a + \lambda_c).$$

6. DATA ACQUISITION & REDUCTION

Having constructed models that allowed comparison of the FTS techniques, the paths that the FTS/imp took during execution had to be monitored to supply the raw data to determine the transitional probabilities.

[4]In the 2-state model, there is no recovery when a failure occurs. The recovery process in figure 5.1 is part of the processing to 1 of the 4 output arcs in figure 5.2. Thus, the model in figure 5.2 traces every injected fault to its final outcome. Figure 5.3 further collapses the outcomes into Good & Failed states.

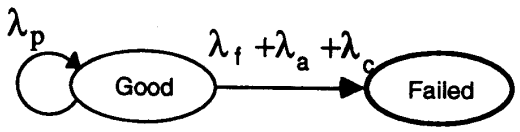

Figure 5.3. Continuous-Time Markov Model

6.1 *Event Path Monitoring*

Event-paths were monitored by inserting software sensors into the FTS/imp to identify when each of the following events occurred during program execution:

- an error (active fault) was detected,
- a recovery was complete,
- an FTS/imp run was aborted.

These events could be used to directly identify certain state transitions; *ie*, the error-detected message indicates the transition from Error → Detected, and the run-aborted message indicates a transition from Detected → Abort. Referring to figure 5.1 (GDMM), a recovery-complete message could indicate either Detected → Recovered for a successful recovery, or Detected → Error for an unsuccessful recovery.

The identification of additional states had to come from fault-injection and testing of the FTS/imp. Since the faults were injected dynamically within a test harness, the time at which the fault was encountered was recorded, as well as whether or not the final results produced by the execution of the FTS/imp matched a *golden standard*. The *golden standard* was the base/imp that had passed over 30k test-vectors, as described in section 1. A result of the FTS/imp's matching the *golden standard* indicated a transition into Pass, while a mismatch indicated a transition into Fail.

An abnormal end of a test execution or an abrupt end of a test trace was interpreted as a transition to Crash. An abrupt end of data indicated that the test environment itself had crashed. Although FTS/imp executions with this type of crash could be attributed to external factors, they were included in the analysis based on the assumption that a severe crash by the FTS/imp could very well cause a complete crash of the testing environment.

The detection of a special type of event was termed a "lucky" completion. This occurred when a correct result was obtained (*eg*, Pass), but from incorrect CMV values, indicating that the fault was masked by the compression action of the specification. When such a completion occurred, it was assumed the program was previously in the Error state. This type of completion was the critical factor in determining if: arriving in the Recovery Complete state indicated a —

Detected → Recovered → Pass, or

Detected → Error → Pass transition.

6.2 *Event-Path Analysis*

Event characters, as depicted in table 6.1, and corresponding to the states on figure 5.1, were generated by the execution of FTS/imp in conjunction with the test harness. Additional state transitions were synthesized through pattern inferences built into analysis tools. The string of characters produced during execution was termed *event paths*. The output of the FAUST & FIAT parsers produced a string of character files (script files) that contained event paths. In all, over 80k FTS/imp executions generated by various permutations of faults, test-vectors, and testing-environments necessitated the development of automated data-reduction tools to analyze the scripts of event paths.

It was not possible to detect a recovery-started event from the data taken during the experiment. To account for this, an assumption was made that an error-detected event led to a recovery-started event. Hence in all event paths, a *d* event is always followed by an *r* event, even though the *d* is redundant. Event paths can be classified into four categories:

- simple: state transitions made during execution of FTS/imp are obvious from the given sequence of events
- non-standard: paths that contain illegal state-transitions[5]
- ambiguous: paths that can not be applied directly to the model of figure 5.1 and required individual interpretation and translation[6]
- complex: (*recovery loops*) these paths pose a problem in that the internal state of the program for intermediate loops is indeterminate for all loops except those ending in a known state, *eg*, *...drRp* or *...drRf*[7].

The CDMM are generated from the GDMM state-transition counts by the tool COLLAPSE. It uses the iterative matrix-multiply technique to reduce a state-transition probability matrix (generated by normalizing the rows of a transition count matrix) into a set of probabilities of entering 1 of the 4 trapping-states (Abort, Crash, Pass, Fail). These probabilities define the CDMM. COLLAPSE also calculates the MTTFGE as discussed

[5]For example, the path *f*, implies a transition Good → Fail in figure 5.1. This transition not only violates the model but is appropriately unanticipated since the FTS/imp under test was supposedly bug free. Since Good → Fail is a possible (albeit unanticipated) transition, non-standard paths were included in the final state-transition tables for completeness, but ignored in subsequent analyses.

[6]These ambiguous paths necessitated the development of a tool, XLAT, that would isolate the ambiguous string and replace it with the correct interpretation. The output of XLAT was in turn analyzed to generate a state transition matrix by a program BINIT. The main function of BINIT was to compress a list of event paths into a list consisting of only the unique paths, along with the number of times each path was encountered. Direct analysis of the BINIT output was not possible due to the manifestation of complex paths of program execution.

[7]To analyze paths containing recovery loops, the tool REPS was developed to predict the state after a recovery inside a loop. REPS analyses all paths and recovery loops and produces 2 state-transition tables: a) containing the absolute transition counts and totals, and b) giving the 1-step transition probability matrix (transitions probabilities as time → ∞). Many of the results in section 7 are based directly on these tables.

TABLE 6.1
Event-Character Definitions

Symbol	Description	Indicated by
e	Fault encountered (error activated)	Test Harness
d	Error detected	FTS/imp
r	Started recovery process	FTS/imp
R	Completed recovery process	FTS/imp
p	Correct final result (passed)	Test Harness
l	Correct final result from incorrect CMV (lucky)	Parser
f	Incorrect final result (failed)	Test Harness
a	Program aborted gracefully	FTS/imp
c	Program crashed	Parser/Test Harness

in section 5.3. The tool SHARPE [13,15] was used to verify the results of COLLAPSE; the same results were obtained by both programs.

7. RESULTS

This section:

- describes the results that were obtained from the FAUST & FIAT executions;
- describes & compares the figures of merit (see sections 5.1 & 5.2) that were derived from GDMM & CDMM (figures 5.1 & 5.2).

7.1 *CDMM*

CDMM is a simplified model that shows only the reaching of end states of the GDMM without concern for how they are reached. The transition probabilities of the CDMMs for the various FTS/imp (assuming a 100% fault activation rate) are summarized in figures 7.1 & 7.2. These results are used to calculate the correctness & availability (defined in section 5.2) of each FTS/imp.

Figure 7.1 shows the distribution of FAUST-run outcomes (software design fault). In the software design-fault injection method, 207 test vectors with 100 randomly generated faults were injected into the base/imp and FTS/imp. The base/imp represents a non fault-tolerant approach[8]. The 5 possible outcomes are:

- Pass. The CMV and the reported LAUNCH value was correct;
- Lucky. The CMV was incorrect but the reported LAUNCH value was correct;
- Fail. The reported LAUNCH value was incorrect;
- Abort. The FTS/imp detected an error and aborted gracefully;
- Crash. The program either hung or unanticipatedly exited.

The results from the base/imp illustrate that an active fault does not always lead to a failure (Crash or Fail). This error

[8]The program generated by the Software Engineering Group in phase I.

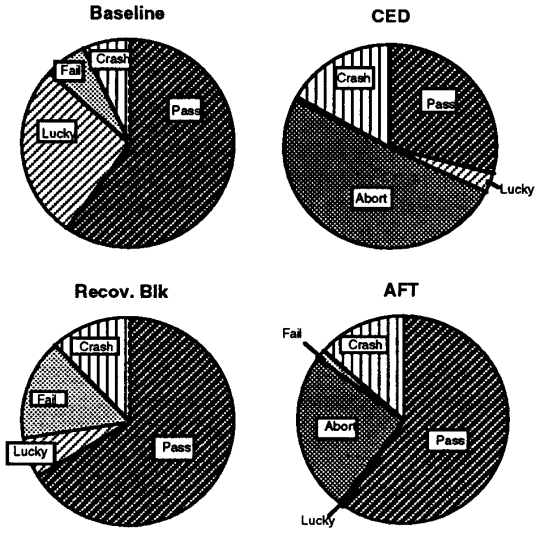

Figure 7.1. Outcome After Injection of a Design Fault

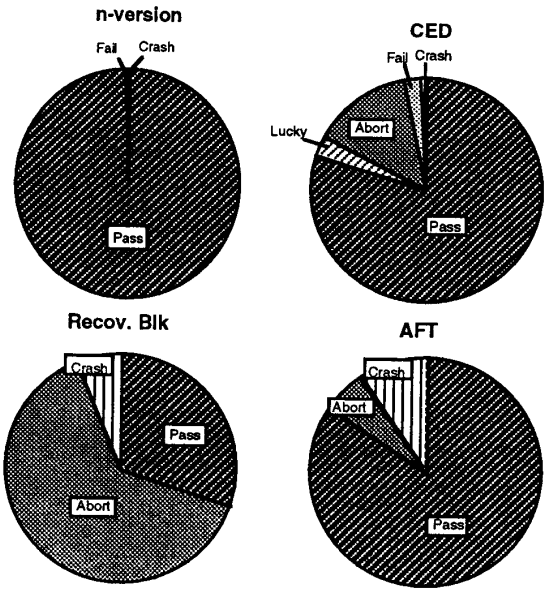

Figure 7.2. Outcome After Injection of a Hardware Fault

masking is particularly pronounced in this experiment due to the nature of the LIP (*eg*, the base/imp passes over 85% of the time in the presence of an active fault). Since we want to be able to draw conclusions which apply to any application, we want to be able to distinguish between success due to error-recovery and to error-masking. Because we know the CMV for each implementation execution, we can determine when an error is masked in the final output (LAUNCH value). We have been describing these events as *lucky* successes.

For the CED FTS/imp, many of the errors that might have been masked are detected, and result in an Abort. All FTS/imp

exhibit a much lower fraction of *lucky* results than the base/imp. We believe this indicates that the FTS/imp are effective at detecting & recovering from errors even though the total fraction of successful outcomes (passes and *lucky* passes) is less for each of the FTS/imp than it is for the base/imp.

Figure 7.1 shows that each FTS/imp is more likely to crash than the base/imp. This is probably due to the extra complexity in the FTS/imp and the fact that the base/imp, unlike the new FTS/imp, was subjected to aggressive testing & validation in phase I.

Figure 7.2 shows the distribution of outcomes when an intermittent fault is injected (via FIAT) to simulate a hardware fault. For FIAT testing, 20 test vectors were run for each of the 100 randomly generated faults. Due to the lack of certain sensor placements in the base/imp, detailed output distinctions (*eg*, Lucky, Pass, Crash, Fail) could not be obtained. Even though there is no base/imp result with which to compare these results, all FTS/imp seem to recover well from a simulated hardware fault. The RB technique was not able to recover from a great majority of the errors; however, aborts are much preferred to *lucky* passes. The N-V FTS/imp, in particular, demonstrates the ability to pass almost 100% of the time. Error masking is even more prevalent for intermittent faults, as evidenced by the ability of the CED FTS/imp (which has no recovery mechanism) to pass a great majority of the time.

7.1.1 MTTFGE

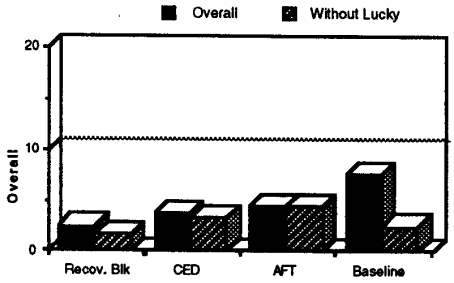

Figure 7.3. MTTFGE Comparison

Using the definition of MTTFGE developed in section 5.3, we can calculate the MTTFGE for each FTS/imp. Figure 7.3 shows the MTTFGE for 3 of the FTS/imp as well as the base/imp. Due to lack of data for the N-V FTS/imp (only data for hardware fault injections to the code section) it was not included in the graph. An MTTFGE is also given under the assumption that a *lucky* completion is counted as a failure (to factor-out error masking). The base/imp seems to outperform all other FTS/imp until the *lucky* cases are factored out. Since the *lucky* cases represent only a small fraction of the error masking, we suspect that in an application less prone to error masking, a non-fault-tolerant approach would not perform as well. On the other hand, there is a relatively small change in the MTTFGE of the other FTS/imp when *lucky* cases are factored out. This indicates that their fault-tolerance relies on error masking to a lesser extent. Using only data from hardware fault injections to the code section of the N-V FTS/imp (20% of FIAT fault injections) the MTTFGE was a factor of 100 higher than the others. However, this cannot be a fair comparison as only 20% of FIAT data and no FAUST data are considered.

7.1.2 Summary

FTO	Error Detection	Error Recovery	Aborting	Correctness	Availability	MTTF	MTTF w/o Luck
n-version	5	1	—	1	1	1	1
Recov. Blk	2	4	3	5	4	4	5
CED	4	—	1	2	4	4	4
AFT	1	2	3	2	3	4	4
Baseline	—	—	—	5	5	3	5

1 2 3 4 5
better ◄——► worse

Figure 7.4. Overall FTS Implementation Comparison

Figure 7.4 summarizes how well each of the FTS/imp performed with respect to the figures of merit that were identified. From section 5.2, correctness is $(1-P_f)$ and availability is (P_p+P_a).

Judging from figure 7.4, the N-V FTS/imp easily outperforms the others. Care should be taken, however, in drawing too strong a conclusion from the N-V FTS/imp results since:

- some important fault classes (design & hardware-data faults) were not considered.
- the same number of faults were injected into all FTS/imp even though N-V FTS/imp is by far the longest (table 9.1). This gives the N-V FTS/imp an advantage in low fault density.

Among the FTS/imp for which the data were complete, the AFT FTS/imp seems to have better reliability & availability than the other alternatives. One might anticipate CED FTS technique to perform well in correctness & availability due to its relative simplicity (no recovery overhead). The CED FTS/imp did perform well in detecting most errors. Its apparent inability to detect hardware data errors, however, resulted in its mediocre overall error-detection rate.

We can only suppose that the apparent poor performance of the RB FTS/imp is due to the complexity of (and, possibly, the presence of bugs in) its acceptance test. In retrospect, the application considered in this experiment (LIP) did not lend itself well to the RB FTS technique due to the absence of a simple acceptance test. By contrast, if the application had, for example,

been sorting, then an acceptance test that was much less complex than the original task (pairwise comparison of adjacent items in the list) would have been possible.

The RB FTS is vulnerable in its acceptance test and recovery mechanism. If faults are injected into these critical sections, the recovery mechanism is damaged — making error recovery less likely.

Even though the data for the N-V FTS/imp were not complete, it is still reasonable to ask what characteristics of the N-V FTS technique were responsible for its apparent success. We have identified 3 such aspects:

- high level of redundancy (reduced the single-point of failure to one module - the controller),
- high quality of the redundant versions (the implementations are from validated LIPs of phase I),
- ability to use error masking (CMV values are voted-on as well).

7.2 *GDMM*

This section analyzes data collected on probabilities pertaining to certain arcs and paths in the GDMM of figure 5.1. These data were gathered through sensors in the FTS/imp to flag fault execution, error detection, recovery after detection, and aborts. The N-V data are not included in figures 7.5 - 7.7 since, a) no FAUST testing was performed, and b) FIAT data were incomplete. For FIAT fault injections, the sensors in the N-V FTS/imp were enabled only when code (and not data) portions were being tested. This was due to some misunderstanding of testing specifications. All of the figures represent percentage of faults that were executed (as shown in figures 4.1 & 4.2), not of all faults injected.

7.2.1 Probability of Error Detection

The Pr{an error is detected | a fault is active} can be found directly from the GDMM (figure 5.1) for the FTS as the probability of Error → Detect. Figures 7.5a & 7.5b show the probability of error detection exhibited by each of the FTS/imp for hardware (FIAT) and design (FAUST) errors, respectively. Figure 7.5c shows the average detection probability. As pointed out in section 4.1.1, no design-error detection data is available for the N-V FTS/imp. However, 79% of the hardware fault injections to the code portion of the N-V FTS/imp were detected. While the N-V FTS technique does not provide any provisions for explicit error detection, the N-V FTS/imp considered any discrepancies in the voters as error detection. This is true for our particular experiment since all versions were confirmed to work properly with the test sets that were used.

In summary, an average of 40-60% of the errors were detected by all 4 FTS/imp. There were, however, some interesting characteristics in their distribution. The RB FTS/imp is effective in detecting errors in the data area while the CED FTS/imp is good at detecting code errors. This is reasonable since the CED FTS technique monitors only the execution path of the target program. The AFT FTS/imp seems to be very effective in detecting design errors, and shows a high detection probability average over both hardware & design errors.

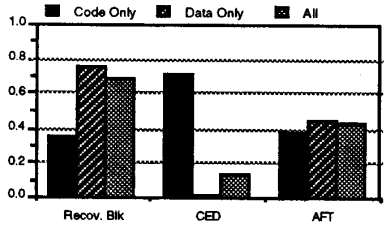

Figure 7.5a. Probability of Error Detection (physical/hardware faults)

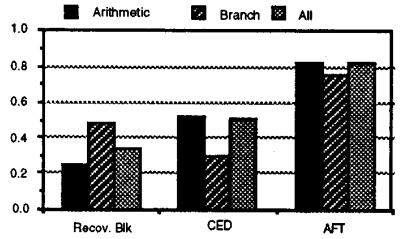

Figure 7.5b. Probability of Error Detection (design faults)

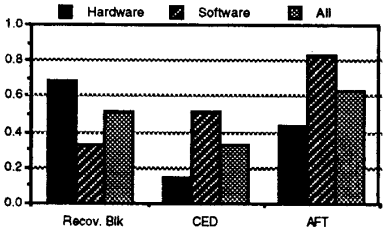

Figure 7.5c. Probability of Error Detection (Average)

7.2.2 Probability of Error Recovery

The probability of error recovery can also be read directly from the GDMM for each FTS/imp. It is Pr{making the detected → recovered transition}. Figures 7.6a - 7.6c show the probability of error recovery for the FTS/imp.

The CED FTS/imp scheme has no recovery mechanism. The N-V FTS technique was able to recover from all FIAT injected faults to the code portion of the FTS/imp. Figure 7.6c shows that while the AFT FTS/imp is able to recover equally well from hardware & design faults, the RB FTS/imp recovers from hardware faults much less often than from software design faults.

7.2.3 Probability of Aborting

The final figure of merit derived from the GDMM is the probability of aborting after error detection given that successful

recovery is not possible (the alternative is faulty or incomplete recovery). The probability of aborting is calculated from the GDMM by:

(Detect → Abort)/[(Detect → Abort) + (Detect → Error)].

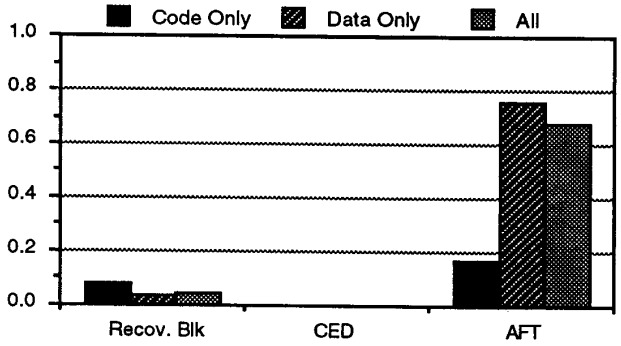

Figure 7.6a. Pr{Error Recovery after Detection} (physical/hardware faults)

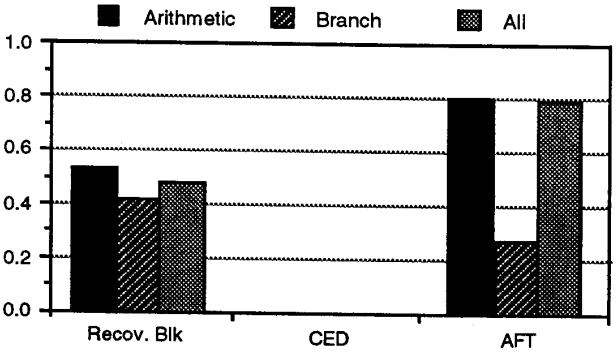

Figure 7.6b. Pr{Error Recovery after Detection} (design faults)

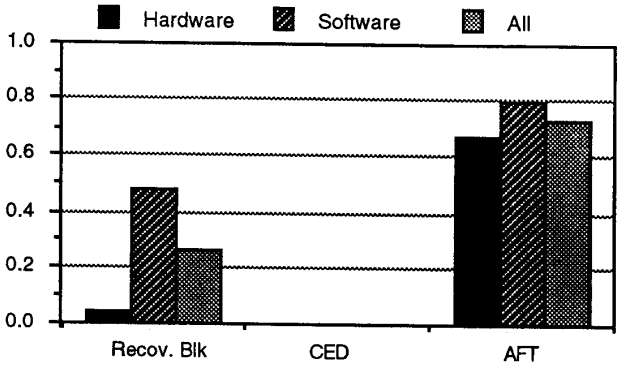

Figure 7.6c. Pr{Error Recovery after Detection} (Average)

Figure 7.7 shows this probability for each of the FTS/imp. No data was available on the N-V FTS/imp since it was always able to recover successfully. The CED FTS/imp also represents a special case since it never attempts to recover. The RB FTS/imp always aborts (rather than recover incorrectly) from hardware data errors. The same is true for the AFT FTS/imp for software control flow errors.

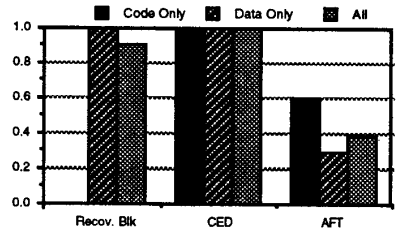

Figure 7.7a. Pr{Aborting} (physical/hardware faults).

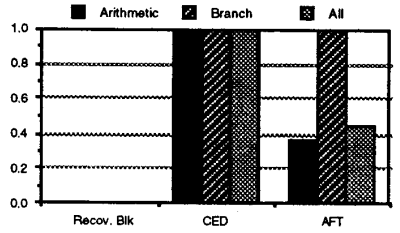

Figure 7.7b. Pr{Aborting} (design faults).

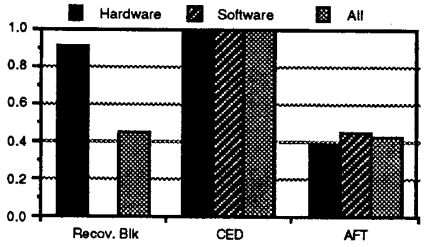

Figure 7.7c. Pr{Aborting} (Average)

8. VARIATION OF THE ERROR RATE

Error Rate ≡ (number of errors encountered)/execution. Since our analysis assumed one injected-fault and bug-free FTS/imp, the error rate is simply:

Pr{encountering the error in a single execution},

which is given by the,

Good → Error arc in the GDMM (figure 5.1).

All the figures of merit in section 7 are normalized to an error rate of 1.0. In an actual implementation, the Pr{encountering an error} can vary over a wide range. As Pr{encountering an

error} increases, then Pr{arriving in an unsuccessful state[9]} increases. If all other transition probabilities remain the same, the failure-probability varies linearly with Pr{Good → Error}.

The Pr{encountering a single design error} can be estimated from the test data. For example, when design faults were injected into the RB FTS/imp, they were encountered only 33.5% of the times. Table 8.1 gives the results for the 3 FTS/imp and the baseline software.

Some interesting observations can be made after examining table 8.1. For example, the probability of encountering a single design fault is much smaller with RB FTS/imp than with any other FTS/imp. This can be understood by noting that if the fault is injected in a backup recovery code, it is never encountered. With AFT FTS/imp on the other hand, the fault detection and recovery code is more interleaved with normal code, resulting in the higher error rate for the AFT FTS/imp.

TABLE 8.1
Probability of Encountering a Single Design Error

Algorithmic Fault-Tolerance	78.9%
Concurrent Error- Detection	51.2%
Recovery Block	33.5%
Baseline	62.7%

9. RANKING FTS TECHNIQUES FOR VARIOUS APPLICATIONS

This section proposes a further abstraction of modeling data and gives the results of applying it to this experiment. An obvious goal of this type of experiment is to conclude how good each method of fault tolerance is relative not only to the others, but to a base/imp as well. Each FTS technique has strengths & weaknesses. While the idea of abstracting data to a few points of comparison is attractive, to reduce the performance of each FTS/imp to a single benchmark is overly optimistic; too much information is lost in the simplification. On the other hand, some generalization is needed in order to draw conclusions. Therefore, step 1 in a meaningful comparison is to introduce domains in which rankings make sense.

9.1 *Reliability Comparisons*

Different systems have different needs in terms of reliability. The data from this experiment show that the different approaches to fault tolerance meet these needs to varying degrees. Our objective, therefore, is to develop a metric for each application-type that could be used to evaluate & compare the effectiveness of fault tolerance techniques. For a given application, some of the ways an FTS technique can terminate (*eg*, Abort, Crash, Fail, Pass) are more or less desirable relative to the others. The metric should therefore consider the FTS technique's tendency to reach a given state in the case of a fault. Essentially, the metric should weight individual state probabilities differently based on the factors important to that application.

1. A suitable application taxonomy is needed. The 4-class taxonomy [17] was chosen because:

- the factors important to each type of system are clearly defined,
- the taxonomy seems sufficient to be of practical use,
- the factors seem to map easily onto the failure modes of our model.

The 4 application types are:

- general purpose,
- long life,
- high availability,
- critical computation.

Four corresponding metrics are therefore needed and are derived here.

1. *General Purpose* (GP). Typified by commercial systems. It is important that the system be in a good or recovered state most of the time. So, the comparison metric must consider the probability that the system is up as well as producing correct output.

$$GP = 1 - P_c - P_f - P_a.$$

2. *High Availability* (HA). Systems that require low downtime. If a function is carried out, such as a committed transaction in a banking system, the results must be valid. If an unrecoverable error is encountered, though, a quick abort is the most desirable outcome (rather than incorrect output).

$$HA = 1 - P_c - P_f.$$

3. *Long Life* (LL). Systems where maintenance is impossible. An example is unmanned spacecraft which are either very expensive or impossible to repair after launch. So, the most important failure to be avoided is crashing. If a failure occurs, one would definitely prefer a graceful exit such as an abort or even an occasional incorrect result over a catastrophic failure.

$$LL = 1 - P_c.$$

4. *Critical Computation* (CC). Real-time control systems where each failure represents a critical loss. In this experiment, all 3 of the failure modes must be avoided at all costs. The mean time-to-failure, given a fault, offers just such a value.

$$CC = 1/(P_c + P_f + P_a).$$

For various reasons, systems can be more susceptible to either hardware faults or to design faults. A system designer might, for example, want to focus on tolerating hardware errors because they are the most anticipated, so the designer should choose a method for fault tolerance that best handles that error

[9]Failed, Crashed, or Aborted.

class. A system's susceptibility to a hardware or design faults can vary over time. If this progression can be anticipated or detected, it might be advantageous to "progress" the fault tolerance technique as well.

To allow for different situations, the FAUST data (modeling design fault tolerance) was given differing weights with respect to the FIAT data (hardware fault tolerance). Table 9.1 presents the probabilities for two specific cases:

- design fault tolerance (DFT) weighted 80% and hardware fault tolerance (HFT) 20%,
- vice versa.

TABLE 9.1
Ranking Metrics (Application *vs* FTS Implementation)
[The table-body is the measure stated in the first part of this section. The top line in each group is for weights: DFT,HFT → 80%,20%. The bottom line in each group is for weights: DFT,HFT → 20%,80%. Some groups have only 1 line.]

	Baseline*	AFT	CED	N-V†	RB
GP $(1-P_{cfa})$	0.869	0.623	0.388		0.628
		0.685	0.609	0.927	0.334
HA $(1-P_{cf})$	0.869	0.835	0.820		0.748
		0.785	0.807	0.927	0.844
LL $(1-P_c)$	0.933	0.875	0.855		0.891
		0.897	0.934	0.931	0.922
CC $1/P_{cfa}$	7.6	2.69	1.77		3.19
		3.22	2.68	262	1.79

*The FIAT data are unavailable; DFT,HFT → 100%,0%
†The FAUST data are unavailable; DFT,HFT → 0%,100%
DFT = design-fault tolerance
HFT = hardware-fault tolerance
$P_{cf} = P_c \cdot P_f$; $P_{cfa} = P_c \cdot P_f \cdot P_a$

An extraction of the rankings by selecting the 3 largest numbers in each row, done once for each DFT,HFT weighting, yields table 9.2. (A 50%,50% weighting was calculated in each table but is not shown.) The apparent success of the base/imp over several FTS/imp is interesting; it ranks first in 3 of 12 cases. However, this is attributable to the limitations of our experiment; the base/imp has an advantage in that more stringent *tolerance* criteria were imposed on the FTS/imp than was imposed on the base/imp. The FTS/imp detect the internal errors that are most likely masked in the baseline, and then the FTS/imp either attempt-recovery or abort, rather than risk-producing-an-incorrect-answer; this behavior is advantageous especially in a program that produces an analog result.

9.2 *Cost Comparisons*

This research has focused on the reliability of the 4 FTS/imp. There are, however, some costs related mainly to implementation that must be considered when assessing the applicability of an FTS. These can include:

- time required to develop the FTS/imp,
- personnel required to develop FTS/imp,
- execution speed,
- CPU and data-storage requirements,
- complexity of testing environment,
- maintenance.

TABLE 9.2
FTS Implementation Comparison
[The body of the table gives the 3 best applications for each group. See table 9.1 for data. The column headings are weights: DFT,HFT (see table 9.1).]

	80%,20%	20%,80%	50%,50%
GP	N-V†	N-V†	N-V†
	Baseline*	CED	Baseline*
	AFT	AFT	AFT
HA	N-V†	N-V†	N-V†
	Baseline*	AFT	CED
	AFT	RB	AFT
LL	Baseline*	CED	Baseline*
	N-V†	Baseline*	N-V†
	RB	N-V†	RB
CC	N-V†	N-V†	Baseline*
	Baseline*	Baseline*	N- V†
	AFT	AFT	AFT

*,† See footnotes on table 9.1

We did not measure or evaluate any of the above characteristics, however we can provide some insight into a related issue, *viz*, code size. Table 9.3 lists the number of source lines of code (SLOC) for the FTS/imp. The SLOC counting program included lines in the .h files. Comments were not counted. The CED, AFT, and N-V FTS/imp all used a common communication module in the test harness (LINKS), which contributed 546 SLOC to each of their totals. Code that was replicated, such as the N-V voters and version shells, was counted only once.

TABLE 9.3
FTS Implementation Source Lines of Code

FTS Implementation	SLOC
Concurrent Error-Detection	3933
Recovery Block	3650
Algorithmic Fault- tolerance	3473
N-Version	5116
Baseline	1060

From table 9.3: All of the FTS/imp had, at a minimum, approximately 3.4 times the SLOC as the base/imp. Relating this to implementation considerations implies that 3 to 4 times as many programmers are required to produce a FTS/imp solution (assuming the productivity of each person is the same).

From figure 7.7: The overall comparison of the FTS techniques indicates that N-V yielded high marks, in addition to having the largest SLOC. The other FTS techniques produced approximately the same number of SLOC.

Using this information, in conjunction with the application taxonomy presented earlier, can help to determine the best FTS for an application, as well as the associated development effort.

10. OBSERVATIONS AND LESSONS LEARNED

At the base of the analysis was the GDMM. A major issue of this model is the nebulous definition of *state*. As discussed in section 5.1, it is difficult to define clearly the *state* of a software program. This definition is a tradeoff among the competing factors of feasibility, accuracy, and generality. We based our definition of *state* on time-based events that could be directly measured. It is apparent from the development of the REPS tool, however, that this method did not provide enough internal state data to determine probabilities such as successful recovery adequately. It was not *infeasible* to extract greater state information from each FTS/imp (since an individual model would have to be constructed for each FTS/imp); rather it was *undesirable* lest the comparisons become too finely tuned to this particular experiment. The ambiguities were accepted for the sake of a common model that allowed some overall comparisons.

A multiple-error model is necessary to model these FTS techniques more accurately. We assumed that the FTS/imp were bug-free, yet a very few illegal state transitions (*eg*, Good → Fail) were encountered, indicating that our assumption was not completely valid. A multiple-error model is most desirable in that it would not only account for these testing anomalies, but could better predict the future performance of FTS with multiple bugs still present in the code at release time, as determined by models such as Musa [10]. The attempt in section 8 to describe the dependence of the MTTFGE on $\Pr\{G \rightarrow E\}$ makes the case for a multiple error model even more salient.

An interesting and somewhat unanticipated result is that the added complexity of an FTS/imp can be self-defeating. This was evidenced by the better performance of the base/imp as described in section 9. This is not surprising in retrospect, however, because in some FTS techniques, such as AFT or RB, the code devoted to FTS/imp was at least as complicated as the basic LIP (base/imp).

We discovered that the lack of error visibility in the LIP made it difficult to evaluate the performance of the error recovery mechanisms accurately. In interpreting event paths, we encountered ambiguity resulting from not knowing if a recovery attempt had been successful. In analyzing the various figures of merit for the FTS/imp it was difficult and sometimes impossible to determine if a test passed because of successful recovery or because of error masking. Error visibility could have been improved in one of two ways:

- by selecting an application that did not inherently mask errors a great deal,
- by requiring the FTS/imp to report more of their state (we only required them to report the final value of the CMV & LAUNCH).

We found that judging from the ability of the baseline (non fault-tolerance) and the N-V FTS technique to make the most of error masking, there is an interesting tradeoff when selecting & implementing an approach to fault-tolerance. On the one hand, it seems desirable to detect all errors early, before they can propagate and make recovery difficult. However, our results show that (at least in our experiments) errors can often have no effect on the final result or even on internal state. Since error states often *die out* there might be an advantage to letting them develop, and attempting to repair only those that are likely to affect the final result. The N-V FTS technique used this approach to some extent since it attempted to repair an error state only when the error propagated to the result of a higher-level subroutine (a CMV value calculating routine in one of the versions), in a sense, taking advantage of the fault tolerance inherent in the application. The CED FTS technique, by contrast, attempted to detect errors aggressively before their results propagated to the result of a CMV value calculating routine, thus causing an abort when the error might have died out if no action had been taken. The best choice of an FTS technique, then, seems to be one which can detect errors early enough to repair them efficiently while still taking advantage of any fault-tolerance inherent in the application.

ACKNOWLEDGMENT

We are pleased to acknowledge the efforts of all 27 participants in the class that designed & performed the experiments that generated the data which formed the basis of this paper. The course participants responded to the challenge, far exceeding the requirements of a typical graduate course. Only space limitations prevent listing all of the participants. We are also pleased to thank the anonymous referees that provided helpful comments in the paper's early stages.

REFERENCES

[1] T. Anderson *et al*, "Software fault tolerance: An evaluation", *IEEE Trans. Software Engineering*, vol SE-11, 1985 Dec, pp 1502–1510.

[2] A. Avizienis, "The *n*-version approach to fault-tolerant software", *IEEE Trans. Software Engineering*, vol SE-11, 1985 Dec, pp 1491–1501.

[3] D. Eckhart, *et al*, "An experimental evaluation of software redundancy as a strategy for improving reliability", *IEEE Trans. Software Engineering*, vol 17, 1991 Jul, pp 692–702.

[4] J. Gray, "A census of Tandem system availability between 1985 and 1990", *IEEE Trans. Reliability*, vol 39, 1990 Oct, pp 409–418.

[5] K-H. Huang, J. A. Abraham, "Algorithm-based fault tolerance for matrix operations", *IEEE Trans. Computers*, vol C-33, 1984 Jun, pp 518–528.

[6] L. R. Kleinrock, *Queuing Systems*, 1977; Prentice Hall.

[7] J. C. Knight, N. G. Leveson, "An experimental evaluation of the assumptions of independence in multiversion programming", *IEEE Trans. Software Engineering*, vol SE-12, 1986 Jan, pp 96–109.

[8] H. J. Larson, B. O. Shubert, *Probabilistic Models In Engineering Sciences*, vol I & II, 1979; John Wiley & Sons.

[9] A. Mahmood, E. J. McCluskey, "Concurrent error detection using watchdog processors - A survey", *IEEE Trans. Computers*, vol 37, 1988 Feb, pp 160–174.

[10] J. D. Musa, *Software Reliability: Measurement, Prediction, Application*, 1990; McGraw-Hill.

[11] B. Randell, "System structure for software fault-tolerance", *IEEE Trans. Software Engineering*, vol SE-1, 1975 Jun, pp 220–232.
[12] T. R. N. Rao, E. Fujiwara, "Error-control coding for computer systems", 1989; Prentice Hall.
[13] R. Sahner, K. Trivedi, "Reliability modeling using SHARPE", *IEEE Trans. Reliability*, vol R-36, 1987 Jun, pp 186–192.
[14] Segall, Barton, Vrsalovic, Siewiorek, Dancey, Robinson, "FIAT - Fault injection based automated testing environment", *IEEE Proc. 18th Int'l Symp. Fault-Tolerant Computing*, 1988 Jun 30.
[15] "SHARPE-Language Description Manual", 1991 Feb; Electrical Engineering Department, Duke University.
[16] D. P. Siewiorek, R. S. Swarz, *The Theory and Practice of Reliable System Design*, 1982; Digital Press.
[17] D. P. Siewiorek, "Architecture of fault-tolerant computers: An historical perspective", *Proc. IEEE*, vol 79, 1991 Dec, pp 1710–1734.
[18] B. Suh, C. Fineman, Z. D. Segall, "FAUST - Fault injection based automated software testing", *Proc. 1991 Systems Design Synthesis Technology Workshop*, 1991 Sep 10–13; NSWC, Silver Spring.
[19] D. J. Taylor, D. E. Morgan, J. P. Black, "Redundancy in data structures: improving software fault tolerance", *IEEE Trans. Software Engineering*, vol SE-6, 1980 Nov, pp 585–594.
[20] D. J. Taylor, D. E. Morgan, J. P. Black, "Redundancy in data structures: Some theoretical results", *IEEE Trans. Software Engineering*, vol SE-6, 1990 Nov, pp 595–602.
[21] K. Trivedi, R. Giest, "Reliability estimation of fault-tolerant systems: tools and techniques", Computer, 1990 Jul, pp 52–61.

AUTHORS

John J. Hudak; Computer Engineering Center; Carnegie Mellon University; Pittsburgh, Pennsylvania 15213-3890 USA.

John J. Hudak received his BSEET degree in Electrical Engineering from the University of Pittsburgh at Johnstown in 1975, and his MSEE from Carnegie Mellon University in 1983. He is pursuing PhD work at CMU. He is a Principal Scientist at Carnegie Mellon Research Institute, Computer Engineering Center. His research interests include high performance computer architecture, fault tolerant systems, and real-time systems. He is a member of IEEE Computer Society, Association for Computing Machinery, and a registered Professional Engineer in the state of Pennsylvania.

Byung-Hoon B. Suh; Electrical & Computer Engineering Dept.; Carnegie Mellon University; Pittsburgh, Pennsylvania 15213-3890 USA.

Byung-Hoon B. Suh received the BS & MS in Electrical & Computer Engineering from Carnegie Mellon University in 1985 & 1987. He is a PhD candidate in the Electrical & Computer Engineering Department. His research interests include software testing and verification, fault tolerant computing, and computer performance analysis.

Dr. Zary Segall; Electrical & Computer Engineering Dept; Carnegie Mellon University; Pittsburgh, Pennsylvania 15213-3890 USA.

Zary Segall received the MSc & PhD in Computer Science from Technion, Israel Institute of Technology in 1976 & 1979. He is an Associate Professor of Computer Engineering and Computer Science. His research interests are in the testing & validation of highly dependable real-time distributed systems, and parallel processing and parallel computers as a way of achieving high performance and increasing reliability over a rich range of algorithms. His research includes the development of theoretical & practical technology for performance efficient parallel programming. Dr. Segall is the Co-Director of the Center For Dependable Systems (CDS).

Dr. Daniel P. Siewiorek; Electrical & Computer Engineering Dept.; Carnegie Mellon University; Pittsburgh, Pennsylvania 15213-3890 USA.

Daniel P. Siewiorek received the BS in Electrical Engineering from the University of Michigan, Ann Arbor in 1968, and the MS & PhD in Electrical Engineering (minor in Computer Science) from Stanford University in 1969 & 1972. Dr. Siewiorek is a Professor in the School of Computer Science and the CIT Department of Electrical & Computer Engineering at Carnegie Mellon University, where he helped to initiate and guide the Cm* project that culminated in an operational 50-processor multiprocessor system. He has been a key contributor to the reliability & maintainability of over 24 commercial systems. His research interests include computer architecture, reliability modeling, fault-tolerant computing, modular design, and design automation. He has conducted research and served as a consultant to several commercial & government organizations, while serving on 6 technology advisory committees. Dr. Siewiorek has written textbooks on parallel processing, computer architecture, reliable computing, and design automation, and over 275 papers. Elected an IEEE Fellow in 1981 for contributions to the design of modular computing systems, he received the IEEE Computer Society and the Association for Computing Machinery Eckert-Mauchly Award in 1988 for his outstanding contributions in parallel computer architecture, reliability, and computer architecture education. Dr. Siewiorek is Director of the CMU Engineering Design Research Center Design for Manufacturing Laboratory. He is a member of the IEEE, ACM, Tau Beta Pi, Eta Kappa Nu, and Sigma Xi.

Manuscript TR92-303 received 1992 May 12; revised 1992 December 1.

IEEE Log Number 07621 ◄TR►