International Journal of Research (IJR) Vol-1, Issue-7, August 2014 **ISSN 2348-6848**

A Smart Software Testing Tool

M. Rajasekar

Research Scholar, Hindustan University

sekarca07@gmail.com

ABSTRACT:

Software testing is all about finding defects in applications. It's nearly impossible to test software under all of the conditions it will run in, and even more difficult to understand how an application will react if the execution environment suddenly becomes hostile or catastrophic. tests the applications in hostile environment. In that kind of environment we don't know the result of following problems such as network card failure, memory leak, insufficient memory, library file corruption or file missing etc. Smart Software Testing Tool (SSTT) will simulate every kind of problem that may occur when application runs. It will have different kind of add-on modules to test an application in all applicable scenarios. If software does not experience any problems during execution then it cannot behave badly only when it encounters problems that corrupt its program state can things go away. The usefulness in the defect modeling and building fault tolerant software systems are not properly preached and/or practiced. There are various types of fault injection. In this paper I have discussed about fault injection to generate environment which a software tester can't. It will generate Database and File related fault injection, which enhance software tolerance against the faults. We can improvise the productivity of software application.

Keywords: *Software Testing Tool, Software Testing, Software Testing Tool (SSTT), software application*

1. INTRODUCTION:

1.1 OVERVIEW OF SOFTWARE TESTING

Software Testing is the process of executing a program or system with the intent of finding errors. Or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible.

Unlike most physical systems, most of the defects in software are design errors, not manufacturing defects. Software does not suffer from corrosion, wear and tear generally it will not change until upgrades, or until obsolescence. So once the software is shipped, the design defects or bugs will be buried in and remain latent until activation.

Discovering the design defects in software, is equally difficult, for the same reason of complexity. Because software and any digital systems are not continuous, testing boundary values are not sufficient to guarantee correctness. All the possible values need to be tested and verified, but complete testing is infeasible. Exhaustively testing a simple program to add only two integer inputs of 32-bits (yielding 2^64 distinct test cases) would take hundreds of years, even if tests were performed at a rate of thousands per second. Obviously, for a

realistic software module, the complexity can be far beyond the example mentioned here. If inputs from the real world are involved, the problem will get worse, because timing and unpredictable environmental effects and human interactions are all possible input parameters under consideration.

1.2 TO IMPROVE SOFTWARE QUALITY

As computers and software are used in critical applications, the outcome of a bug can be severe. Bugs can cause huge losses. Bugs in critical systems have caused airplane crashes, allowed space shuttle missions to go awry, halted trading on the stock market, and worse. Bugs can kill. Bugs can cause disasters. The so-called year 2000 (Y2K) bug has given birth to a cottage industry of consultants and programming tools dedicated to making sure the modern world doesn't come to a screeching halt on the first day of the next century. In a computerized embedded world, the quality and reliability of software is a matter of life and death.

Quality means the conformance to the specified design requirement. Being correct, the minimum requirement of quality, means performing as required under specified circumstances. Debugging, a narrow view of software testing, is performed heavily to find out design defects by the programmer. The imperfection of human nature makes it almost impossible to make a moderately complex program correct the first time. Finding the problems and get them fixed, is the purpose of debugging in programming phase.

1.2 SMART SOFTWARE TESTING TOOL (SSTT)

Smart Software Testing Tool (SSTT) integrates with various features to simulate scenarios which a tester cannot. Such as network latency, disk error, disk latency, dll issues, memory leaks, insufficient memory, file corrupt etc. IST tool have the capability to include new scenario modules.

Expose sensitive data that hackers can exploit. Crash applications to expose software failures missed by exception handlers. Generates the reports based on the types of scenarios tested. Can avoid security vulnerabilities and other sensitive data expose to real world. Component Based Service provides multi-level testing.

2. LITERATURE REVIEW:

The literature review consolidates the understanding on fault injection, associated topics and subsequent studies to emphasis the need to fault injections in business software application. It also crystallizes the need for awareness, tools and analyzes defect leakage/amplification. Even after 20 years of existence the awareness of fault injection and associated modeling with tools are very rarely used and understood in the commercial software industry and used. The usefulness in the defect modeling and building fault tolerant software systems are not properly preached and/or practiced. Added, the availability of appropriate literature and software tools is very few and not used in commercial and business application design and testing.

2.1 RELATED WORKS IN FAULT INJECTION

In recent years there has been much interest in the field of software reliability and fault tolerance of systems and commercial software. This in turn has resulted in a wealth of literature being published around the topic, such as the Fault Injection in the form of the ‘Marrying Software Fault Injection Technology Results with Software Reliability’ by Jeffrey Voas, Digital Norman Schneidewind. Many critical business computer applications require “fault tolerance," the ability to recover from errors or exceptional conditions. Error free

software is very difficult to create and creating fault tolerant software is an even greater challenge. Fault tolerant software must successfully recover from a multitude of error conditions to prevent harmful system failures. Software testing cannot demonstrate that a software system is completely correct. An enormous number of possible executions that must be examined in any real world sized software system. Fault tolerance expands the number of states (and thus execution histories) that must be tested, because inconsistent or erroneous states must also be tested. Mailing lists, websites, research and forums have been created in which all aspects of this fresh new niche software engineering area are discussed. People are interested, partly because it is a new area but also because the whole field of commercial software reliability is in itself so interesting; as it holds so many wide ranging disciplines, perspectives and logic at its core. Software reliability engineering is uniting professionals in disciplines that previously had little to do with one another, it is creating more opportunities for employment in the online environment, and it is changing the face and structure of all information that we seek out on the web. In the era of economic recession, customer demands reliable, certified and fault tolerant commercial and business software applications.

In this research, the focus is on software testing techniques that use fault injection. Several potentially powerful existing systems have drawbacks for practical application. We first examine existing fault injection techniques and evaluate their potential for practical application in commercial and business software applications. Available and accessible literature infrastructure including premium subscribed IEEE and ACM resources were studied and summarized for literature review from 1986 (20 years).

2.3 FAULT INJECTION

2.3.1. Fault Injection Really

The main problem with fault injection is to know what to do with it. Upon first glance, it would seem to be a good tool for debugging a system, and detecting any flaws within it. Once one examines the procedures and the information gained, however, it becomes apparent that fault injection is good at testing known sorts of bugs or defects, but poor at testing novel faults or defects, which are precisely the sorts of defects we would want to discover. Therefore, what emerges is that fault injection is not really suited for debugging and improving the system so much as it is suited for testing the fault tolerant features of the system. A known fault in injected and the results examined to see if the system can respond correctly despite the fault.

2.3.2. Uses of Fault Injection

Along these lines, there are two proposed uses for fault injection. The first is for verification of a system. If a system is designed to tolerate a certain class of faults, or exhibit certain behavior in the presence of certain faults, then these faults can be directly injected into the system to examine their effects. The system will either behave appropriately or not, and it's fault tolerance measured accordingly. For certain classes of ultra-dependable un-testable systems in which the occurrence of errors is too infrequent to effectively test the system in the field, fault injection can be a powerful tool for accelerating the occurrence of faults in the system and verifying that the system works properly.

The other proposed use for fault injection is less well understood, because the problem it addresses is poorly understood. Robustness is used in regard to systems these days almost synonymously with fault tolerance, but robustness actually embraces more than this. There is no really good definition of robustness, but it is something along the lines of "the capability

of a system to behave correctly in unusual conditions." The difficulty lies in creating unusual conditions so as to test the system for robustness. Fault injection has been proposed as a method to address this problem, by including unusual conditions as well as faults. This would provide us with a metric for measuring the robustness of a system.

2.3.3. Difficulties in Fault Injection

There are two difficulties that must be addressed before this use of fault injection can be fully applied. The first is the disparate nature of systems, and the ways in which they can fail or experience faults. Unless two systems are set to accomplish the exact same task, determining the relative robustness of the two systems is a difficult task. A good metric for robustness would be able to resolve this difference. Secondly, it is not yet certain how our metric should be biased. Common practice is to have the test distribution mirror the real world distributions of occurrence of faults. If we are truly testing the system's response to unusual situations, however, it might be better to bias the test towards the less frequently encountered conditions. While there is agreement that fault injection can serve as a metric for robustness, the exact mechanisms of doing so are as of yet poorly understood.

2.3.4. SOFTWARE FAULT INJECTION

Software fault injection is used to inject faults into the operation of software and examine the effects. This is generally used on code that has communicative or cooperative functions so that there is enough interaction to make fault injection useful. All sorts of faults may be injected, from register and memory faults, to dropped or replicated network packets, to erroneous error conditions and flags. These faults may be injected into simulations of complex systems where the interactions are understood though not the details of implementation, or they may be injected into operating systems to examine the effects.

Software simulations are typically of high level description of a system, in which the protocols or interactions are known, but not the details of implementation. These faults tend to be mis-timings, missing messages, replays, or other faults in communication in a system. The simulation is then run to discover the effects of the faults. Because of the abstract nature of these simulations, they may be run at a faster speed that the actual system might, but would not necessarily capture the timing aspects of the final system. This sort of testing would be performed to verify a protocol, or to examine the resistance of an interaction to faults.

This would typically be done early in the design cycle so as to flesh out the higher level details before attempting the task of implementation. These simulations are non-intrusive, as they are simulated, but they may not capture the exact behavior of the system.

Software fault injections are more oriented towards implementation details, and can address program state as well as communication and interactions. Faults are mis-timings, missing messages, replays, corrupted memory or registers, faulty disk reads, and almost any other state the hardware provides access to. The system is then run with the fault to examine its behavior. These simulations tend to take longer because they encapsulate all of the operation and detail of the system, but they will more accurately capture the timing aspects of the system. This testing is performed to verify the system's reaction to introduced faults and catalog the faults successfully dealt with. This is done later in the design cycle to show performance for a final or near-final design. These simulations can be non-intrusive, especially if timing is not a concern, but if timing is at

all involved the time required for the injection mechanism to inject the faults can disrupt the activity of the system, and cause timing results that are not representative of the system without the fault injection mechanism deployed. This occurs because the injection mechanism runs on the same system as the software being tested.

2.3.5. HARDWARE FAULT INJECTION

Hardware fault injection is used to inject faults into hardware and examine the effects. Typically this is performed on VLSI circuits at the transistor level, because these circuits are complex enough to warrant characterization through fault injection rather than a performance range, and because these are the best understood basic faults in such circuits. Transistors are typically given stuck-at, bridging, or transient faults, and the results examined in the operation of the circuit. Such faults may be injected in software simulations of the circuits, or into production circuits cut from the wafer.

Hardware simulations typically occur in a high level description of the circuit. This high level description is turned into a transistor level description of the circuit, and faults are injected into the circuit. Typically these are stuck-at or bridging faults, as software simulation is most often used to detect the response to manufacturing defects. The system is then simulated to evaluate the response of the circuit to that particular fault. Since this is a simulation, a new fault can then be easily injected, and the simulation re-run to gauge the response to the new fault. This consumes time to construct the model, insert the faults, and then simulate the circuit, but modifications in the circuit are easier to make than later in the design cycle. This sort of testing would be used to check a circuit early in the design cycle. These simulations are non-intrusive, since the simulation functions normally other than the introduction of the fault.

Hardware fault injections occur in actual examples of the circuit after fabrication. The circuit is subjected to some sort of interference to produce the fault, and the resulting behavior is examined. So far, this has been done with transient faults, as the difficulty and expense of introducing stuck-at and bridging faults in the circuit has not been overcome. The circuit is attached to a testing apparatus which operates it and examines the behavior after the fault is injected. This consumes time to prepare the circuit and test it, but such tests generally proceed faster than simulation does. It is, rather obviously, used to test circuit just before or in production. These simulations are non-intrusive, since they do not alter the behavior of the circuit other than to introduce the fault. Should special circuitry be included to cause or simulate faults in the finished circuit, these would most likely affect the timing or other characteristics of the circuit, and therefore be intrusive.

2.3.6. FAULT INJECTION MODELLING

Fault Injection Modeling (FIM) involves the deliberate insertion of faults or errors into a computer system in order to determine its response. It has proven to be an effective method for measuring and studying response of defects, validating fault-tolerant systems, and observing how systems behave in the presence of faults. In this study, faults are injected in all phases of Software Development Life Cycle viz., Requirements, Design and Source code.

2.3.7. Objectives of Fault Injection

The objectives of conducting these experiments are to measure process efficiencies, statistically study, analyze and report defect amplification of defects (Domino's effect) across software development phases with a similar system constructed with technological variation.

The goal of this research is to understand the behavior of faults and defects pattern in commercial and business software application and defect leakage in each phase of application development. Throughout the literature certain questions reoccur, which one would anticipate when a new field emerges in commercial software fault tolerance. People are interested, and want to understand and define commercial software reliability and fault tolerance, so the following questions which are recurrent throughout the literature are not surprising:

- Why study Fault Injection Modeling?
- Why study business software fault tolerance requirements?
- Why are they called 'Fault Injection & Error Seeding'?
- Why review Software Implemented Fault Injection (SWIFI)?
- What work was performed, current status and work proposed?

These questions will be expanded upon throughout the research, and seek to bring clarity to those who want to find the answers to the above, or to see if there truly are any answers.

2.3.8. Domino's effect

Domino's effect is the cascading effect of defects from the initial stages of the project to all the subsequent stages of the software life cycle. Errors undetected in one work product are 'leaked' to the child work product and *amplifies* defects in the child work product. This chain reaction causes an exponential defect leakage. E.g.: undetected errors in requirements leak and cause a significant number of defects in design which, in turn, causes more defects in the source code. The result of this study is to arrive at an "Amplification Index" which will characterize the extent of impact or damage of phase-wise defects in subsequent Software Development Life Cycle (SDLC) phases.

3. PROBLEM DESCRIPTION & SOLUTION:

3.1. OVERVIEW

Fault injection requires the selection of a fault model [5]. The choice of this model depends on the nature of faults. Software errors arising from hardware faults, for instance, are often modeled via bits of zeroes and ones written into a data structure or a portion of the memory [15, 21], while protocol implementation errors arising from communication are often modeled via message dropping, duplication, reordering, delaying etc. [14]. Understanding the nature of security faults provides a basis for the application of fault injection. Several studies have been concerned with the nature of security faults [1, 3, 6, 16, 20].) However, we are not aware of any study that classifies security flaws from the point of view of environment perturbation. Some general fault models have also been widely used [13, 21]. The semantic gap between these models and the environment faults that lead to security violations is wide and the relationship between faults injected and faults leading to security violations is not known.

We have developed an Environment-Application Interaction (EAI) fault model which serves as the basis the fault injection technique described here. The advantage of the EAI model is in its capability of emulating environment faults that are likely to cause security violations. Another issue in fault injection technique is the location, within the system under test, where faults are to be injected. In certain cases, the location is obvious. For example, in [14], the faults emulated are communication faults. Hence, the communication channels between communicating entities provide the obvious location for fault injection. In other cases, where the location is hard to decide, nondeterministic methods, such as random selection, selection according to

distribution, are used to choose the locations. The selection of location is also a major issue for us. In the current stage of our research, we inject environment faults at the points where the environment and the application interact. In future work, we plan to exploit static analysis to further reduce the number of fault injection locations by finding the equivalence relationship among those locations. The motivation for using static analysis method is that we can reduce the testing efforts by utilizing static information from the program. A general issue about software testing is "what is an acceptable test adequacy criterion?" [10]. we adopt a two-dimensional coverage metric (code coverage and fault coverage) to measure test adequacy.

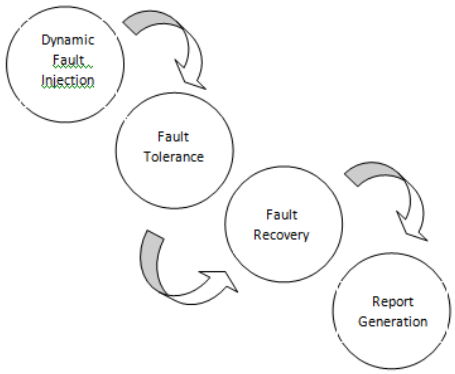
3.2 SOLUTION

There is a plethora of testing methods and testing techniques, serving multiple purposes in different life cycle phases. Classified by purpose, software testing can be divided into: correctness testing, performance testing, reliability testing and security testing. Classified by life-cycle phase, software testing can be classified into the following categories: requirements phase testing, design phase testing, program phase testing, evaluating test results, installation phase testing, acceptance testing and maintenance testing. By scope, software testing can be categorized as follows: unit testing, component testing, integration testing, and system testing. But all kind of testing evaluates the application, no other testing method used to test the application in different kind of environments.

To make an impeccable software quality we need to test the application according to the hostile environments which will result in software tolerance and increase in productivity. For that reason we have to develop and follow environment centric software testing procedures. Plethora of ways to test an application but testing a software in a hostile environment will improvise the software quality by many folds and secures sensitive data safe.

4. ARCHITECTURE:

Smart Software Testing Tool consist following modules as shown below,



4.1. DYNAMIC FAULT INJECTION

Integrated with Dynamic Fault Injection. It is achieved by Component Based Software System. Bunch of faults related to specific category will be created as Add-on Component.

For instance for network may consist of following scenarios time-out, error in connection, network unreachable, network card- hardware failure, etc. These faulty scenarios may come into a single Network Add-on for Dynamic Fault Injection. Using this every kind of fault scenario we can simulate and test at once.

4.1.1. EXCEPTION HANDLING: TYPES OF PROBLEM

- **C**omputational problem
- **H**ardware problem
- **I** /O and file problems
- **L**ibrary function problem
- **D**ata input problem
- **R**eturn-value problem: function or procedure call

- **E**xternal user/client problem
- **N**ull pointer and memory problems

4.1.2. PROGRAMMED EXCEPTION HANDLING

Programmed exception handling modules are mechanisms built into software for specific exceptional cases that are known are likely to occur. Since these occurrences are relatively well understood, protection for them can be incorporated into the system. When a program is executing, if one of the exceptional conditions is detected, control is passed from the main process block to the special exception handling block. This code will deviate from normal execution to compensate for the exceptional condition and will attempt to mask it to prevent propagating an error condition to higher levels in the software hierarchy.

If the condition cannot be recovered, the exception handler may call check pointing recovery code to return the system to a known state before the exception occurrence and retry the operation.

4.2. FAULT TOLERANCE

Fault tolerance is one of the most important means to avoid service failure in the presence of faults, so to guarantee they will not interrupt the service delivery. Software testing, instead, is one of the major fault removal techniques, realized in order to detect and remove software faults during software development so that they will not be present in the final product.

4.2.1 Fault Tolerance Engineering: from Requirements to Code

In the past, fault tolerance (and specifically, exception handling) used to be commonly delayed until late in the design and implementation phases of the software life-cycle. More recently, however, the need for explicit use of exception handling mechanisms during the entire life cycle has been advocated by some researchers as one of the main approaches to ensuring the overall system dependability.

It has been recognized, in particular, that different classes of faults, errors and failures can be identified during different phases of software development. A number of studies have been conducted so far to investigate where and how fault tolerance can be integrated in the software life-cycle. In the remaining part of Section 4 we will show how fault tolerance has been recently addressed in the different phases of the software process: requirements, high-level (architectural) design, and low-level design.

4.2.2 Requirements Engineering and Fault Tolerance

Requirements engineering is concerned with identifying the purpose of a software system, and the contexts in which it will be used. Various theories and methodologies for finding out, modeling, analyzing, modifying, enhancing and checking software system requirements have been proposed.

4.2.3 Software Fault Tolerance

In this section we present fault tolerance techniques applicable to software. These techniques are divided into two groups: single version and multi-version software techniques. Single version techniques focus on improving the fault tolerance of a single piece of software by adding mechanisms into the design targeting the detection, containment, and handling of errors caused by the activation of design faults. Multi-version fault tolerance techniques use multiple versions (or variants) of a piece of software in a structured way to ensure that design faults in one version do not cause system failures. A characteristic of the software fault tolerance techniques is that they can, in principle, be applied at any level in a software system: procedure, process, full application program, or the whole system including the operating system. Also, the

techniques can be applied selectively to those components deemed most like to have design faults due to their complexity.

4.2.4 Single-Version Software Fault Tolerance Techniques

Single-version fault tolerance is based on the use of redundancy applied to a single version of a piece of software to detect and recover from faults. Among others, single-version software fault tolerance techniques include considerations on program structure and actions, error detection, exception handling, checkpoint and restart, process pairs, and data diversity.

4.2.5 Software Structure and Actions

The software architecture provides the basis for implementation of fault tolerance. The use of modularizing techniques to decompose a problem into manageable components is as important to the efficient application of fault tolerance as it is to the design of a system. The modular decomposition of a design should consider built-in protections to keep aberrant component behavior in one module from propagating to other modules. Control hierarchy issues like visibility (i.e., the set of components that may be invoked directly and indirectly by a particular component [2]) and connectivity (i.e., the set of components that may be invoked directly or used by a given component [2]) should be considered in the context of error propagation for their potential to enable uncontrolled corruption of the system state.

Partitioning is a technique for providing isolation between functionally independent modules [3]. Partitioning can be performed in the horizontal and vertical dimensions of the modular hierarchy of the software architecture [2]. Horizontal partitioning separates the major software functions into highly independent structural branches communicating through interfaces to control modules whose function is to coordinate communication and execution of the functions. Vertical partitioning (or factoring) focuses distributing the control and processing work in a top-down hierarchy, where high level modules tend to focus on control functions and low level modules do most of the processing. Advantages of using partitioning in a design include simplified testing, easier maintenance, and lower propagation of side effects [2].

System closure is a fault tolerance principle stating that no action is permissible unless explicitly authorized [4]. Under the guidance of this principle, no system element is granted any more capability than is needed to perform its function, and any restrictions must be expressly removed before a particular capability can be used. The rationale for system closure is that it is easier (and safer) to handle errors by limiting their chances of propagating and creating more damage before being detected. In a closed environment all the interactions are known and visible, and this simplifies the task of positioning and developing error detection checks. With system closure, any capability damaged by errors only disables a valid action. In a system with relaxed control over allowable capabilities, a damaged capability can result in the execution of undesirable actions and unexpected interference between components.

Temporal structuring of the activity between interacting structural modules is also important for fault tolerance. An atomic action among a group of components is an activity in which the components interact exclusively with each other and there is no interaction with the rest of the system for the duration of the activity [12]. Within an atomic action, the participating components neither import nor export any type of information from other non-participating components. From the perspective of the non-participating components, all the activity within the atomic action appears as one and indivisible

occurring instantaneously at any time during the duration of the action. The advantage of using atomic actions in defining the interaction between system components is that they provide a framework for error confinement and recovery. There are only two possible outcomes of an atomic action: either it terminates normally or it is aborted upon error detection. If an atomic action terminates normally, its results are complete and committed. If a failure is detected during an atomic action, it is known beforehand that only the participating components can be affected. Thus error confinement is defined (and need not be diagnosed) and recovery is limited to the participating set of components.

4.3. Error Detection

Effective application of fault tolerance techniques in single version systems requires that the structural modules have two basic properties: self-protection and self-checking [1]. The self-protection property means that a component must be able to protect itself from external contamination by detecting errors in the information passed to it by other interacting components. Self-checking means that a component must be able to detect internal errors and take appropriate actions to prevent the propagation of those errors to other components. The degree (and coverage) to which error detection mechanisms are used in a design is determined by the cost of the additional redundancy and the run-time overhead. Note that the fault tolerance redundancy is not intended to contribute to system functionality but rather to the quality of the product. Similarly, detection mechanisms detract from system performance. Actual usage of fault tolerance in a design is based on trade-offs of functionality, performance, complexity, and safety.

Anderson [12] has proposed a classification of error detection checks, some of which can be chosen for the implementation of the module properties mentioned above. The location of the checks can be within the modules or at their outputs, as needed. The checks include replication, timing, reversal, coding, reasonableness, and structural checks.

- Replication checks make use of matching components with error detection based on comparison of their outputs. This is applicable to multi-version software fault tolerance discussed in section.

- Timing checks are applicable to systems and modules whose specifications include timing constraints, including deadlines. Based on these constraints, checks can be developed to look for deviations from the acceptable module behavior. Watchdog timers are a type of timing check with general applicability that can be used to monitor for satisfactory behavior and detect "lost or locked out" components.

- Reversal checks use the output of a module to compute the corresponding inputs based on the function of the module. An error is detected if the computed inputs do not match the actual inputs. Reversal checks are applicable to modules whose inverse computation is relatively straightforward.

- Coding checks use redundancy in the representation of information with fixed relationships between the actual and the redundant information. Error detection is based on checking those relationships before and after operations. Checksums are a type of coding check. Similarly, many techniques developed for hardware (e.g., Hamming, M- out-of-N, cyclic codes) can be used in software, especially in cases where the information is supposed to be merely referenced or transported by a module

from one point to another without changing its contents. Many arithmetic operations preserve some particular properties between the actual and redundant information, and can thus enable the use of this type of check to detect errors in their execution.

- Reasonableness checks use known semantic properties of data (e.g., range, rate of change, and sequence) to detect errors. These properties can be based on the requirements or the particular design of a module.

- Structural checks use known properties of data structures. For example, lists, queues, and trees can be inspected for number of elements in the structure, their links and pointers, and any other particular information that could be articulated. Structural checks could be made more effective by augmenting data structures with redundant structural data like extra pointers, embedded counts of the number of items on a particular structure, and individual identifiers for all the items ([5], [6], [10], [11]).

Another fault detection tool is run-time checks [7]. These are provided as standard error detection mechanisms in hardware systems (e.g., divide by zero, overflow, and underflow). Although they are not application specific, they do represent an effective means of detecting design errors.

Error detection strategies can be developed in an ad-hoc fashion or using structured methodologies. Ad-hoc strategies can be used by experienced designers guided by their judgment to identify the types of checks and their location needed to achieve a high degree of error coverage. A problem with this approach stems from the nature of software design faults. It is impossible to anticipate all the faults (and their generated errors) in a module. In fact, according to [1]:

"If one had a list of anticipated design faults, it makes much more sense to eliminate those faults during design reviews than to add features to the system to tolerate those faults after deployment. The problem, of course, is that it is unanticipated design faults that one would really like to tolerate."

Fault trees have been proposed as a design aid in the development of fault detection strategies [8]. Fault trees can be used to identify general classes of failures and conditions that can trigger those failures. Fault trees represent a top-down approach which, although not guaranteeing complete coverage, is very helpful in documenting assumptions, simplifying design reviews, identifying omissions, and allowing the designer to visualize component interactions and their consequences through structured graphical means. Fault trees enable the designer to perform qualitative analysis of the complexity and degree of independence in the error checks of a proposed fault tolerance strategy. In general, as a fault tree is elaborated, the structuring of the tree goes from high-level functional concepts to more design dependent elements. Therefore, by means of a fault tree a designer can "tune" a fault detection strategy trading-off independence and requirements emphasis on the tests (by staying with relatively shallow and mostly functional fault trees) versus ease of development of the tests (by moving deeper down the design structure and creating tests that target particular aspects of the design).

4.4. CONSIDERATIONS ON THE USE OF THECKPOINTING

We are concerned in this section with the use of check pointing during execution of a program. The results referenced here assume instantaneous detection of errors from the moment a fault is activated. In real systems these detection delays are non-zero and should be taken into account when selecting a check pointing strategy. Non-zero detection delays can invalidate checkpoints if the time to detect errors is larger than the interval between checkpoints.

As mentioned above, there exist two kinds of check pointing that can be used with the checkpoint and restart technique: static and dynamic check pointing. Static checkpoints take single snapshots of the state at the beginning of a program or module execution. With this approach, the system returns to the beginning of that module when an error is detected and restarts execution all over again. This basic approach to check pointing provides a generic capability to recover from errors that appear during execution. The use of the single static checkpoint strategy allows the use of error detection checks placed at the output of the module without necessarily having to embed checks in the code. A problem with this approach is that under the presence of random faults, the expected time to complete the execution grows exponentially with the processing requirement. Nevertheless, because of the overhead associated with the use of checkpoints (e.g., creating the checkpoints, reloading checkpoints, restarting), the single checkpoint approach is the most effective when the processing requirement is relatively small.

Dynamic check pointing is aimed at reducing the execution time for large processing requirements in the presence of random faults by saving the state information at intermediate points during the execution. In general, with dynamic check pointing it is possible to achieve a linear increase in actual execution time as the processing requirements grow. Because of the overhead associated with check pointing and restart, there exist an optimal number of checkpoints that optimizes a certain performance measure. Factors that influence the check pointing performance include the execution requirement, the fault tolerance overhead (i.e., error detection checks, creating checkpoints, recovery, etc.), the fault activation rate, and the interval between checkpoints. Because checkpoints are created dynamically during processing, the error detection checks must be embedded in the code and executed before the checkpoints are created. This increases the effectiveness of the checks and the likelihood that the checkpoints are valid and usable upon error detection.

[15] presents three basic dynamic check pointing strategies: equidistant, modular, and random. Equidistant check pointing uses a deterministic fixed time between checkpoints. [15] shows that for an arbitrary duration between equidistant checkpoints, the expected execution time increases linearly as the processing requirement grows. The optimal time between checkpoints that minimizes the total execution time is shown to be directly dependent on the fault rate and independent of the processing requirements.

Modular check-pointing is the placement of checkpoints at the end of the sub-modular components of a piece of software right after the error detection checks for each sub-module are complete. Assuming a component with a fixed number of sub-modules, the expected execution time is directly related to the processing distribution of the sub-modules (i.e., the processing time between

checkpoints). For a given failure rate, a linear dependence between the execution time and the processing requirement is achieved when the processing distribution is the same throughout the modules. For the more general case of a variable processing requirement and an exponential distribution in the duration of the sub-modules, the execution time becomes a linear function of the processing requirements when the check pointing rate is larger than the failure rate.

In random check-pointing the process of checkpoint creation is triggered at random without consideration of the status of the software execution. Here it is found that the optimal average check pointing rate is directly dependent on the failure rate and independent of the processing requirements. With this optimal check pointing rate, the execution time is linearly dependent on the processing requirement.

4.5. MULTI-VERSION SOFTWARE FAULT TOLERANCE TECHNIQUES

Multi-version fault tolerance is based on the use of two or more versions (or "variants") of a piece of software, executed either in sequence or in parallel. The versions are used as alternatives (with a separate means of error detection), in pairs (to implement detection by replication checks) or in larger groups (to enable masking through voting). The rationale for the use of multiple versions is the expectation that components built differently (i.e, different designers, different algorithms, different design tools, etc) should fail differently [16]. Therefore, if one version fails on a particular input, at least one of the alternate versions should be able to provide an appropriate output. This section covers some of these "design diversity" approaches to software reliability and safety.

4.5.1 Recovery Blocks

The Recovery Blocks technique ([17]) combines the basics of the checkpoint and restart approach with multiple versions of a software component such that a different version is tried after an error is detected (Figure 9). Checkpoints are created before a version executes. Checkpoints are needed to recover the state after a version fails to provide a valid operational starting point for the next version if an error is detected. The acceptance test need not be an output-only test and can be implemented by various embedded checks to increase the effectiveness of the error detection. Also, because the primary version will be executed successfully most of the time, the alternates could be designed to provide degraded performance in some sense (e.g., by computing values to a lesser accuracy). Like data diversity, the output of the alternates could be designed to be equivalent to that of the primary, with the definition of equivalence being application dependent. Actual execution of the multiple versions can be sequential or in parallel depending on the available processing capability and performance requirements. If all the alternates are tried unsuccessfully, the component must raise an exception to communicate to the rest of the system its failure to complete its function. Note that such a failure occurrence does not imply a permanent failure of the component, which may be reusable after changes in its inputs or state. The possibility of coincident faults is the source of much controversy concerning all the multi-version software fault tolerance techniques.

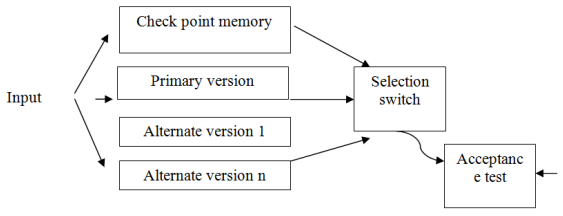


Figure 9: Recovery Block model

4.5.2 N-Version Programming

N-Version programming [18] is a multi-version technique in which all the versions are designed to satisfy the same basic requirements and the decision of output correctness is based on the comparison of all the outputs (Figure 10). The use of a generic decision algorithm (usually a voter) to select the correct output is the fundamental difference of this approach from the Recovery Blocks approach, which requires an application dependent acceptance test. Since all the versions are built to satisfy the same requirements, the use of N- version programming requires considerable development effort but the complexity (i.e., development difficulty) is not necessarily much greater than the inherent complexity of building a single version. Design of the voter can be complicated by the need to perform inexact voting. Much research has gone into development of methodologies that increase the likelihood of achieving effective diversity in the final product. Actual execution of the versions can be sequential or in parallel. Sequential execution may require the use of checkpoints to reload the state before an alternate version is executed.

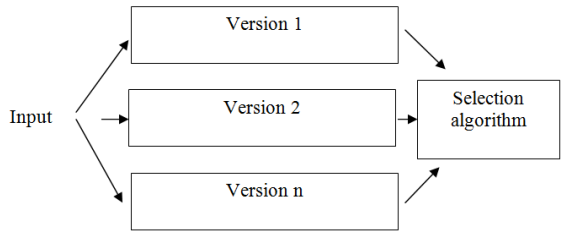


Figure 10: N-Version Programming Model

4.5.3 N Self-Checking Programming

N Self-Checking programming ([19], [20]) is the use of multiple software versions combined with structural variations of the Recovery Blocks and N-Version Programming. N Self-Checking programming using acceptance tests is shown on Figure 11. Here the versions and the acceptance tests are developed independently from common requirements. This use of separate acceptance tests for each version is the main difference of this N Self-Checking model from the Recovery Blocks approach. Similar to Recovery Blocks, execution of the versions and their tests can be done sequentially or in parallel but the output is taken from the highest-ranking version that passes its acceptance test. Sequential execution requires the use of checkpoints, and parallel execution requires the use of input and state consistency algorithms.

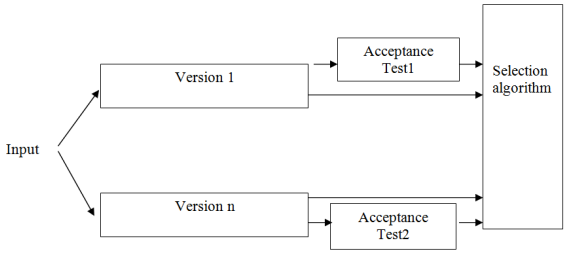


Figure 11: N Self-Checking Programming using Acceptance Tests

4.5.4 Consensus Recovery Blocks

The Consensus Recovery Blocks [21] approach combines N-Version Programming and Recovery Blocks to improve the reliability over that achievable by using just one of the approaches. According to [21], the acceptance tests in the Recovery Blocks suffer from lack of guidelines for their development and a general proneness to design faults due to the inherent difficulty in creating effective tests. The use of voters as in N-Version Programming may not be appropriate in all situations, especially when multiple correct outputs are possible. In that case a voter, for example, would declare a failure in selecting an appropriate output.

Consensus Recovery Blocks uses a decision algorithm similar to N-Version Programming as a first layer of decision. If this first layer declares a failure, a second layer using acceptance tests similar to those used in the Recovery Blocks approach is invoked. Although

obviously much more complex than either of the individual techniques, the reliability models indicate that this combined approach has the potential of producing a more reliable piece of software [21]. The use of the word potential is important here because the added complexity could actually work against the design and result in a less reliable system.

5. IMPLEMENTATION:

Endorses hostile environment software testing. Software development involves with large amount of time and money, if software fails due to an error which could have avoided would become awry. It is becoming common for software engineering standards to enumerate certain classes of problems that must not occur. The argument for doing so is that you can't protect against problems until you know what the problems might be. Here problems are usually defined in one of two ways one is a class of failure that must not occur or the second is a fault class that can occur but the fault class must be shown to only cause acceptable outputs.

In real world there are numerous infamous examples of software failures which shook the industry, incidents like Ariane-5, Therac-25 and so on. The following facets are considered for the implementation of ,

- Error Detection -Detects errors through continuous monitoring,

periodic tests, per-call tests, or other automatic processes. Software audits are considered as part of the error detection capability.

- Error Isolation - Isolates the error to its source, preferably to a

single or a reasonable subset of components.

- Error Recovery- Recovers errors by automatic or manual

actions such as retry, rollback, on-line masking, restart, reload, or re-configuration, to minimize the degradation of service.

- Error Reporting - Sends error messages to a display device, a

logging device, or an Operations System (OS), describing the error, the place where the error is observed, and system reactions to the error.

In our implementation we use storage, message and command based fault injection. Using the storage manipulation tools (for memory, disk, or tape), errors can be injected into the system by changing the value at some location of the storage hierarchy, which represents some system state. Using the commands from the craft interface or remote maintenance terminal, errors can be injected by changing the states of the system entities for operations, administration, maintenance, and provision.

6. RESULT AND DISCUSSION:

Smart Software Testing Tool can be used to improvise the software quality and fault tolerance. When a failure occurs, the system must be able to isolate the failure to the offending component. This requires the addition of dedicated failure detection mechanisms that exist only for the purpose of fault isolation. Recovery from a fault condition requires classifying the fault or failing component.

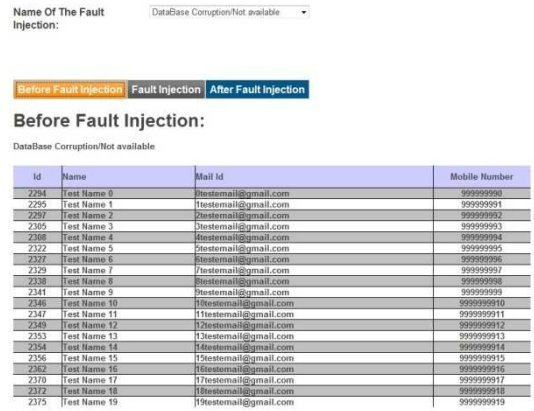
Software fault-tolerance is based more around nullifying programming errors using real-time redundancy, or static "emergency" subprograms to fill in for programs that crash. There are many ways to conduct such fault-regulation, depending on the application and the available hardware. Software fault injection methodology may consume more time than other kinds of testing. But it can avoid software failures in kind of projects like nuclear, space research and so on. These are the areas where Software testing

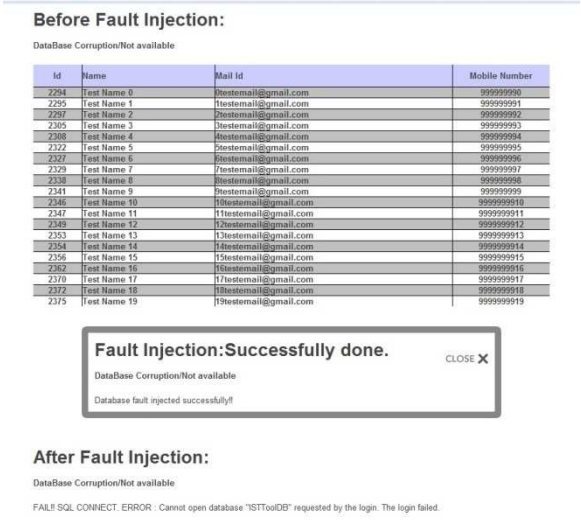
can not be done correctly. When we use fault injection to simulate the environment of the software application, it will reveal potential bugs in the application.

The reason why we need software fault injection and fault tolerance is, it have great capacity to identify the potential bugs happened in real world likens of Ariane-5, Therac-25 and so on. In software application expected functionality not only depends on input anomalies, also includes the external application environments. Ariane 5's first test flight (Ariane 5 Flight 501) on 4 June 1996 failed, with the rocket self-destructing 37 seconds after launch because of a malfunction in the control software. A data conversion from 64-bit floating point value to 16-bit signed integer value to be stored in a variable representing horizontal bias caused a processor trap (operand error) because the floating point value was too large to be represented by a 16-bit signed integer. The software was originally written for the Ariane 4 where efficiency considerations (the computer running the software had an 80% maximum workload requirement) led to 4 variables being protected with a handler while 3 others, including the horizontal bias variable, were left unprotected because it was thought that they were "physically limited or that there was a large margin of error".

An American domestic airlines sold tickets for $1 because of the software failure. And the airlines almost bankrupt due to an bug in software. There are enormous infamous software glitch those can be eliminated using software fault injection and fault tolerance techniques, which would save great amount of time and money.

6.1. SCREENSHOTS





7. CONCLUSION AND FUTURE WORK

This methodology will provide and increase the software quality to the next level. The productivity of the application will be increased. Component based service will help to support more technologies and scenarios related to different types of environment.

Future work will concentrate on applying this methodology to more applications. We are in the progress of developing and conducting a set of experiments to evaluate the effectiveness of this methodology. In the future, we hope to be able to develop a prototype tool for security testing based on this methodology.

8. REFERENCES

[1] Russell J. Abbott, *Resourceful Systems for Fault Tolerance, Reliability, and Safety*, ACM Computing Surveys, Vol. 22, No. 1, March 1990, pp. 35 – 68.

[2] Roger S. Pressman, *Software Engineering: A practitioner's Approach*, The McGraw-Hill Companies, Inc., 1997.

[3] *Software Considerations in Airborne Systems and Equipment Certification*, RTCA/DO-178B,RTCA, Inc, 1992.

[4] Peter J. Denning, *Fault Tolerant Operating Systems*, ACM Computing Surveys, Vol. 8, No. 4, December 1976, pp. 359 - 389.

[5] David J. Taylor, et al, *Redundancy in Data Structures: Improving Software Fault Tolerance*, IEEE Transactions on Software Engineering, Vol. SE-6, No. 6,November 1980, pp. 585 - 594.

[6] David J. Taylor, et al, *Redundancy in Data Structures: Some TheoreticalResults*, IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, November 1980, pp.595 - 602.

[7] Dhiraj K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice-Hall, Inc.,1996.

[8] Herbert Hecht and Myron Hecht, *Fault-Tolerance in Software*, in Fault-Tolerant Computer System Design, Dhiraj K. Pradhan, Prentice Hall, 1996.

[9] IEEE Transactions on Software Engineering, Vol. SE- 12, No. 1, January 1986, pp. 51 – 58.

[10] J. P. Black, et al, *Introduction to Robust Data Structures*, Digest of Papers FTCS-10: The Eleventh Annual International Symposium on Fault Tolerant Computing, October 1 – 3, 1980, pp. 110 - 112.

[11] J. P. Black, et al, *A Compendium of Robust Data Structures*, Digest of Papers FTCS-11: The Eleventh Annual International Symposium on Fault-

Tolerant Computing, June 24 – 26, 1981, pp. 129 – 131.

[12] T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice*, Prentice/Hall, 1981.

[13] Jim Gray, *Why Do Computers Stop and What Can Be Done About It?*, Proceedings of the Fifth Symposium On Reliability in Distributed Software and Database Systems, January 13-15, 1986, pp. 3 - 12.

[14] Paul E. Ammann and John C. Knight, *Data Diversity: An Approach to Software Fault Tolerance*, IEEE Transactions on Computers, Vol. 37, No. 4, April 1988, pp. 418 - 425.

[15] Victor F. Nicola, *Checkpointing and the Modeling of Program Execution Time*, in Software Fault Tolerance, Michael R. Lyu, Ed, Wiley, 1995, pp. 167 – 188.

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

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

[18] Algirdas Avizienis, *The Methodology of N-Version Programming*, in R. Lyu, editor, *Software Fault Tolerance*, John Wiley & Sons, 1995.

[19] Jean-Claude Laprie, et al, *Hardware- and Software-Fault Tolerance: Definition and Analysis of Architectural Solutions*, Digest of Papers FTCS-17: The Seventeenth International Symposium on

Fault-Tolerant Computing, July 1987, pp. 116 - 121.

[20] Jean-Claude Laprie, et al, *Definition and Analysis of Hardware- and Software- Fault-Tolerance Architectures*, IEEE Computer, July 1990, pp. 39 - 51.

[21] R. Keith Scott, James W. Gault, and David F. McAllister, *Fault-Tolerant Software Reliability Modeling*, IEEE Transactions on Software Engineering, Vol. SE-13, No. 5, May 1987, pp. 582 – 592.