Software Fault Tolerance: A Theoretical Overview

Abstract - The progression of software systems implementation and development is always associated with the emergence and evolution of new requirements and services, which in turn will increase the complexity of these systems. It is almost impossible to build error-free software for highly complex systems because it is challenging to detect design errors and to measure the accuracy of the functionality and output for these systems. In some systems such as critical systems, the safety of the software overcomes the reliability of that software, so Software Fault Tolerance (SFT) techniques will be implemented with these systems to avoid catastrophes. The SFT techniques we overview in this paper are divided into dual groups: Traditional Methods and New Techniques. In turn, the traditional techniques are classified into Single-version and Multi-version software fault tolerance techniques, where Single version SFT include techniques such as: “system structuring and closure”, “atomic actions”, “inline fault detection” and “exception handling”. While Multi-version SFT will include methods such as: "Recovery blocks", "N-version programming", "N self-checking programming" and "consensus recovery blocks". On other hand the new techniques will include: "Adaptive n-version Systems", "Fuzzy Voting", "Abstraction", "Rejuvenation" and "Parallel Graph Reduction".

Keywords - Critical Systems, Fault Tolerance, Real Time Systems, Exception Handling, Software Reliability, Software Dependability.

I. INTRODUCTION

Software is widely used in our daily life, but it is very hard and rare to produce “perfect - faultless” software for many reasons.

Usually, bugs in software systems are a life fact [1]. Our computer systems today are suffering from being cruel when it comes to processing faulty software: if an error is founded in the logic, there is an excellent possibility that error is showing up in the autonomous output of the results. The greatest insignificant software problems can only solve without some "trial and error.”

The essence of building software “is the algorithms, data structures, functions, and their interrelationships. Specification, design, and testing of this conceptual construct are the difficult part of software engineering”.

Four embedded possessions make software hard to build: conformity, complexity, invisibility, and changeability. "Software is complex because current software must satisfy a huge quantity of requirements which produces a vast number of states present in the design and the nonlinear interactions among these states. Software is forced to conform because it is perceived as the most conformable of all the components in a system. Software changes continuously because it is extremely flexible. As such, new or updated system requirements (functionality) must be satisfied, consequently is often implemented through software changes. Lastly, the software is invisible. We use programming languages and structures to try to capture the essence of software, but the concepts are so complex that they generally defy attempts to realize them efficiently completely and require the use of techniques to simplify and partition relationships, also enable communication among designing and developing teams” [1, 2].

Program testing could be used for showing the presence of bugs; but, never to explain their absence. Any practical programs are impossible to be tested completely due to the vast number of possible input combinations [3].

The requirement for software to manage with its specific errors, errors initiated by invisible software faults and faults by users seen to have been predictable primarily by designers of real-time control systems [4].

“As we mentioned before, developing fault-free software with total confidence of design correctness is rarely achieved. Hence software fault tolerance techniques are sometimes applied to satisfy design dependability requirements. Software fault tolerance refers to the use of techniques to increase the probability that the final designed system will produce accurate or acceptable (safe) outputs. Since correctness and safety are system level concepts, the need and degree to use software fault tolerance are directly dependent on the characteristics of the intended application and the overall system design”.

II. SOFTWARE DESIGN FAULTS

“ A measure of the success with which the system conforms to some authoritative specification of its behavior when the behavior deviates from that which is specified for it is called a failure” [5].

"Faults and failures are not the same things although the terms are often used fairly loosely. A fault is a static characteristic of a system such as a loose nut on a wheel, an
incorrect statement in a program, an incorrect instruction in
an operational procedure. A failure is some unexpected
system behavior resulting from a fault such as a wheel
falling off or the wrong amount of a chemical being used in
a reactor. Faults can be small and errors can be large” [6, 7,
8].

The presence of error does not necessarily cause a
failure. Failure occurs only if a fault happens during
execution, and a fault occurs only if the program contains a
mistake, but the dual converse statements do not hold, in
general [8, 9].

“Software failures are different from hardware failures
in that software does not wear out. It can continue in
operation even after an incorrect result has been produced.
Current methods of software engineering now allow for the
production of fault-free software. Fault-free software means
software which conforms to its specification. It does NOT
mean software which will always perform correctly as there
may be specification errors” [9, 10].

”Software faults are the root cause in a high percentage
of operating system failures” [2]. It is estimating that 60-
90% of existing computer errors are from software faults

Figure. 1 illustrates the percentage of faults found during
software development processes [12]:

There are four methods for handling software errors:
removal, stopping, fault tolerance, and the sequence of input
workarounds.

“Fault prevention is concentrating on the use of design
methodologies, techniques, and technologies that were
intending to prevent the faults from making a way into the
design”.

“Fault removal concerned with the use of techniques like
reviews, analyses, and testing to check implementation and
remove any faults in that way detected. The proper use of
software engineering during the development processes is a
way of achieving fault prevention and fault removal (i.e.,
fault avoidance)”.  

“Because of the lack of techniques that can ensure that
complex software designs are free of a design fault, fault
tolerance is sometimes used as an extra layer of protection.
Software fault tolerance is the use of techniques to enable
the continued delivery of services at an accurate or
acceptable level of performance and safety after a design
fault becomes active. The selection of particular fault
tolerance techniques depends on system level and software
design specifications and considerations”.

“The input sequence workarounds is used as the last line
of defense against design faults. This is nothing more than
accepting the fact that a particular software design has faults
and taking those faults as features (because, in this instance,
there is no way to” remove these faults from the system
design). This fix (trying to avoid the inputting of data
sequences that may lead to faulty states during the system
operating) is employed by the system operator to work
around known faults while still maintaining the availability
of the system”.

Fault avoidance and fault tolerance are dependability
procurement [13]. So, if not all faults can be found, a
reliable system must be tolerant [12].

III. SOFTWARE RELIABILITY METRICS

There is a number of software engineering
measurements, which termed Software Metrics, to assess
the software quality. But the dual most imperative software
quality ration are safety and reliability.

Reliability is “the probability of failure-free operation of
a computer program in a specified environment for a
specified period”, “where failure-free operation in the
context of software is interpreted as adherence to its
requirements” [2].

Reliability must take precedence over efficiency [14].
To properly consider the reliability of a piece of software,
the impact of a fault must be assessed. “A measure of
software reliability is the Mean Time Between Failures
(MTBF)” as in Eq. 1 [15]:

\[ MTBF = MTTF + MTTR \]

\[ \text{Availability} = \frac{MTTF \times 100\%}{MTTF + MTTR} \]

“Where MTTF is an acronym for the Mean Time To
Failure, and MTTR is the Mean Time To Repair. The
MTTF is a measure of how long a software item is expected
to operate properly before a failure occurs. The MTTR
measures the maintainability of the software (i.e., the degree
of difficulty in repairing the software after a failure
occurs)”.  

Metrics which are employed for identifying software
reliability and availability as follows [17]:

DOI 10.5013/IJSSST.a.20.03.07 7.2 ISSN: 1473-804x online, 1473-8031 print
The concept of reliability of software is useless unless it can be applied to systems with low to moderate reliability. To others, reliability is to deal with the software as a black box. This metric is sometimes called the failure intensity”.

Mean Time To Failure (MTTF): The mediocre time amid perceived system failures. An MTTF of 500 means that one failure can be predictable for each 500-time units.

Availability (AVAIL): The likelihood that the system is available for application at a given time. Availability of 0.998 means that in each 1000 time units, the system is possible to be accessible for 998 of these.

The system's specifications and requirements will determine which metric should be applied to that system. Three classes of measurement are applied to measure the reliability of a system; these measurements are:

1. To measure the POFOD, we can use the number of system malfunctions given a number of requests for system services.
2. To measure the ROCOF and the MTTF, we can use the time (or the number of transactions) between systems failures.
3. To measure the AVAIL, we can use the elapsed reparation or restart time if a system failure happens given that the system should be endlessly accessible.

"What allows the use of reliability as a measure of software quality is the fact that the software is embedded in a stochastic environment that generates input sequences to the software over time. Some of those inputs will result in software failures. Thus, reliability becomes a weighted measure of correctness, with the weights being dependent on the actual use of the software. There is an argument by some software engineers that software reliability has only dual values, which are either 1 (correct) or 0 (incorrect). Therefore, to get any meaningful system reliability estimates, they assume software reliability of 1. To others, the concept of reliability of software is useless unless it can be helpful in reducing the total number of errors in software. An important conclusion of the root cause of the stochastic nature of software failures is that the reliability of a program is not determined by the number of errors in that program but determined by the effect span of these errors. Hence, a program with a high number of errors is not necessarily less reliable than another with a lower number of errors. The only rational approach to estimate software reliability is to deal with the software as a black box. This can be applied to systems with low to moderate reliability requirements".

Figure 2 illustrates the relationship between cost and reliability [17]:

Detecting the cause of software failure (faults) is a good thing. Nonetheless, it's typically much more noteworthy to deal with these faults speedily, safely, and in a deterministic routine. All these features are summarized in a quality called the 'robustness' of the program.

So, robustness can be defined as 'a measure of the ability of a system to be restored from error conditions, whether generated outwardly or on the inside, e.g., a robust system would be easy-going to errors in input data or failures of internal components'. Although there may be a connection amid robustness and reliability, the dual are distinctive measures. A system never requires to be recovered from error conditions may be reliable without being robust. By contrast, an extremely robust system that improves and continues to operate – despite various error conditions – may still be considered as unreliable. The reason? Merely it fails to offer essential services appropriately on demand [18].

Reliability concerns in measuring the probability of failure, ignoring the significances of those failures. In contrast, Software safety concerns in measuring the values of failures from a global system safety perspective.

We can define software system safety as “the software will execute within a system context without contributing to hazards”. A hazard is known as “a state or set of conditions of a system (or an object) that, together with other conditions in the environment of the system (or object), will lead inevitably to an accident (loss event)” [2].

The software fault tolerance techniques aim to embed safety features in the software design or source code to make sure that the software will respond appropriately or an acceptable way to input data errors and prevent output and control errors. The extent and the level of employing error prevention or fault tolerance methods in the software are evaluated by the system requirements and the system safety assessment process. So, the first step for applying a safety feature to system design is by performing modeling and analysis to identify and categorize potential hazards.
Another significant feature in software systems is the dependability. This feature originated from the growing dependence of our society on computing systems associated with the incidence of failures. As mentioned before, most software failures are due to the design faults so that failure can affect one or more attributes of dependability, such as availability or reliability, safety, or confidentiality.

The dependability of a system is that property of a system which allows reliance to be justifiably placed on the service it delivers [19]. In other words, dependability is a quality measure encompassing the concepts of reliability, availability, safety, performability, maintainability, testability and integrity [2, 12, 19, 20].

- **Reliability**: is the probability of a system continues to operate correctly during a particular time interval given that it was operational at the beginning of the interval. A system can be reliable without being fault tolerant, and a system can be fault tolerant without being reliable.
- **Availability**: is the probability that a system is operating correctly at a given time instant.
- **Safety**: "is the probability that the system will perform in a non-hazardous way".
- **Performability**: "is the probability that the system performance will be equal to or greater than some particular level at a given instant of time".
- **Maintainability**: "is the probability that a failed system will be returned to operation within a particular period. Maintainability measures the ease with which a system can be repaired".
- **Testability**: is a measure of the capability to describe a system through testing. Testability consists of the ease of test development (i.e., controllability) and effect observation (i.e., observability).
- **Confidentiality**: the non-occurrence of unauthorized disclosure of information leads to confidentiality.
- **Integrity**: the non-occurrence of improper alterations of information leads to integrity.

To design and develop reliable computing systems, enumerated techniques and methods must be combined and applied for this purpose. These methods and techniques can be characterized into fault prevention, fault tolerance, fault removal, fault forecasting which is how to guess the present number, the future occurrence, and the values of faults.

"The notions introduced up to now can be grouped into three classes and are summarized by” Figure.3.

V. SOFTWARE FAULT TOLERANCE

The goal of applying software fault tolerance techniques to the computing systems design is to ensure that these systems will continue delivering correct or acceptable output although of faults existing in these systems. The reason behind accepting faults in the software is that it is more economical to pay for the consequences of failure rather than detect and remove the faults before system delivery. [21, 22, 23]

From an operational perspective, fault tolerance systems can be classified into dual main classes as shown in Figure. 4. First, there are the ones that defined to be 'fault - tolerant' which must keep going on its operations without any break when faults present (i.e., provide continuous service). Then there are the ones that can go out of operation without employing significant problems (or a problem that can be dealt with), as in the case with a failure an auto-land system at altitude.

Four major subcategories can be derived from the dual classes mentioned above, as follows:

i) **Fail-Operational (FO) system**. This system is designed to give a full performance in the presence of faults. The user and operator will not notice any sign that an error has occurred. e.g., critical systems like the autopilot system in aircraft.

ii) **Fail-Active (FA) system**. This system is designed to give continuous but reduced performance offering acceptable behavior. This behavior is often termed as ‘graceful degradation’. e.g., operating systems in computers.

iii) **Fail-Safe (FS) system**. This system is designed not to break its operation suddenly but must go into safe mode, which is restricted to one of the pre-defined styles, and then need to be manually reset. e.g., network servers.

iv) **High-Availability (HA) system**. This system is designed to return to its full performance after each time its behavior is reduced. This may involve faulty units being replaced while the system is running. e.g., ATM systems.

Fault tolerance actions are [22]:

1) "Fault detection: system determines an incorrect system state has occurred."
2) "Damage assessment: determine system parts affected by fault."
3) "Fault recovery: system must restore its state to a known safe state."

Figure. 3. The dependability tree
4) “Fault repair: for a non-transitory fault, the system is modified to prevent repetition.”

The first step in fault tolerance functionality is to distinguish that a fault happened or the potentiality of its occurrence and it may take an immediate process as the next step to cover that fault. There are dual types of fault detection mechanisms that can be used:

a) “Preventative fault detection. This is the case when the fault detection mechanism is initiated before a state change is committed. If a potentially erroneous state is detected, then the state change is not made.”

b) “Retrospective fault detection. This is the case when the fault detection mechanism is initiated after the system state has been changed to check whether a fault has occurred. If a fault is detected, the fault tolerance system will activate and throw an exception, and then will call a repair mechanism to recover from the fault.”

“Damage assessment involves analyzing the system state to estimate the extent of the state corruption. Damage assessment is needed when you can't avoid making a state change or when a fault is caused by an invalid sequence of individually correct state changes. The role of the damage assessment procedures is not recovered from the fault but to assess what parts of the state space have been affected by the fault”. Damage can only be measured if it is potential to apply several 'validity function' that checks whether the state is consistent. If inconsistencies are found, these are highlighted or signaled in some way.

Fault recovery stands for the process of adapting the state space of the system so that the effects of the fault are removed or lessened. The system can last to operate, perhaps in several degraded forms [24]. When the state of a program is contaminated at run-time, it can be recovered in one of the dual following ways [25]:

a) Either by retrieving a previously saved correct state, and restarting the execution from the point where the state was saved. This process is called backward error recovery.

b) Or by modifying the current (contaminated) state, to obtain a new correct state, then resuming the execution. This process is called forward error recovery.

To achieve a powerful fault-tolerant system design, an intensive and a careful study of a number of items must be done before the design begins, these items are:

- Design.
- Failures.
- Causes of failures.
- System responses to failures.

To measure software fault tolerance, we must [1]:

- “Determine what sections of a software system to test more/less thoroughly than others.”
- “Determine what programs to test more/less thoroughly than others.”
- “Find out what program constructs are more dependable in the presence of bugs (and encourage their use) (e.g., is one loop type more fault tolerant than another?).”
- “Find out what programming languages lead to programs that are more dependable in the presence of bugs and encourage their use) (Are object-oriented languages better, perhaps?).”

“A basic 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 most potential to have design faults due to their complexity”.

 Generally, the demonstrated fault tolerance techniques that can be applied to the software are divided into dual main groups: Traditional techniques and New Techniques. The SFT techniques those included in the previously mentioned dual groups are in there turn classified in many categories, as follows.
A. Traditional Software Fault Tolerance Techniques

The traditional SFT techniques can be categorized into Single version and Multi-version software fault tolerance techniques. Following are some of the most commonly applied classical techniques included in these dual classes.


Redundancy is the key to error detection, correction, and recovery [4]. “Thus Single-version fault tolerance is based on the applying redundancy to a single version of a piece of software (module)”. And so on, single-version software fault tolerance procedures include considerations on program structure and actions, error detection, exception handling, checkpoint and restart, process pairs, and data diversity.

A1.1 Software Structure and Actions: Many techniques focus on software architecture as a basis for applying fault tolerance. The software architecture consists of software structure and actions, which are embedded in many features that facilitating the employment of many fault tolerance techniques, as follows:

- Modularizing Techniques. These techniques are used to dividing the problem (system) into smaller manageable components (modules), in such a way that the design of these components must consider providing built-in protections to prevent the incorrect functionality for one component from propagating to other components.
- The Partitioning Technique. This technique is used to isolate the functionality of independent modules from each other. This will provide the characteristics of simplified testing, easier maintenance, and less propagation of errors that resulted in one module to other modules.
- The System Closure Technique. This technique is used to develop an authority mechanism embedded in the system design. Where this mechanism will not allow the execution of any action unless it is explicitly authorized, so, every component in the system will be granted only the capability that it needs to do its task, and any limitations should be removed before a specific ability can be used. Hence, this technique reduces the possibility of errors propagation between the system components because it provides a more straightforward, safer, and earlier way of detecting and handling errors before they can cause more damages.
- The Temporal Structuring Technique. This technique is based on the concept that the system operations compose from a number of atomic actions. In each atomic action, there is a set of system components in which these components interact exclusively with each other, and there is no interaction with the rest of the other system components. So, the temporal structuring technique will prevent interacted elements from delivering or receiving any data from non-interacted components. This will provide a framework for error isolation. Thus, when a failure is detected during an atomic action, only the interacted components will be affected, and the error will be isolated, and the recovery process will be applied on this set of components.

A1.2 Error Detection: The application of fault tolerance techniques in an effective way in single version software needs that the structured modules of this software to embed dual basic features:

- “The Self-Protection feature. This feature means that a system component must be able to protect itself from external failure infection by detecting the errors in the data passed to it from other interacting components.”
- “The Self-Checking feature. This feature means that a system component must be able to detect the internal errors and take the appropriate actions to prevent the propagation of those errors to other components.”

These dual features are attached to system quality but not to the system functionality and also the detection processes leads to the detraction of the system performance.

There are enumerated types of checks that can be applied within the modules or at their outputs depending on system requirements and design. These checks are as follows:

- Replication checks. These checks are implemented in multi-version software fault tolerance, where the corresponding components (versions) outputs will be compared by a comparison mechanism to detect the occurrence of the error.
- Timing checks. These checks are implemented in systems whose qualifications take account of timing restrictions with deadlines. So, these checks will monitor each system components operations times and detect lost or locked out components according to predefined thresholds. Watchdog timers are a category of timing check.
- Reversal checks. These checks are implemented in systems whose modules inverse computation is relatively linear. Where the outputs of each module are used to calculate the resultant inputs according to the function of these modules. If the computed data do not equal to the real inputs, then an error is detected.
- Coding checks. These checks are implemented in systems by employing "redundancy in the representation of information with fixed relationships between the actual and the redundant information". These checks test those relationships before, and after operations execution, any inconsistency between those relationships will lead to error detection. "Checksums are a type of coding check. Also, many techniques developed for hardware (e.g., Hamming, M-out-of-N, cyclic codes) can be used in software,
particularly in situations where the information is supposed to be just referenced or transported by a module from one point to another without implementing any change on its contents."

- "Reasonableness checks. These checks are implemented in systems by employing well-known semantic features of data (e.g., range, rate of change, and sequence) to detect errors. These features can be based on the necessities and conditions of a particular design of a module."

- Structural checks. “These checks are implemented in systems by employing known properties of data structures. For example, lists, queues, and trees can be tested for the number of elements in the structure, their links and pointers, and any other particular information that could be articulated. Structural checks can be improved to be more powerful by supplying data structures with redundant structural data like additional pointers, embedded counts of the number of items on a particular structure, and individual identifiers for all the items”.

- "Run-Time checks. These checks are implemented usually in hardware systems. These are presented as typical error detection mechanisms in hardware systems (e.g., divide by zero, overflow, and underflow). Although they are not application dependent, they do represent an effective way of detecting design errors".

"Error detection strategies can be developed in an ad-hoc fashion or employing 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. 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 like to tolerate” [2].

One of the vital design aids that have been used in developing fault detecting approaches is Fault Trees. It can be implemented to categorize wide-ranging classes of failures and conditions that can activate those failures. Fault trees are a top-down approach which provides many characteristics, such as it is very useful in recording assumptions, streamlining design reviews, recognizing omissions, and give a visualize representation of component interactions and their concerns through organized graphical means which is very helpful for the designers to present and view their works, and also enable them to accomplish qualitative analysis of the complexity and degree of independence in the error checks of a projected fault tolerance stratagem. But, fault tree is not guaranteeing complete coverage of failures, because it cannot anticipate all classes of failures.

1.3 Exception Handling: The process of interrupting typical sequence of operation to handle abnormal state (operation or output) is termed as exception handling. From the software perspective, when detection mechanisms discover an error, an exception will be thrown, and the system initiates a predefined process to implement a proper recovery.

Although exception handling is one of the most powerful techniques for processing run-time errors, unfortunately, many languages not embed or support the exception construct. In such cases, a workaround is needed to overcome this shortage.

It makes good sense always to define pre- and post-conditions for encapsulated operations. The pre-condition may be translated into the code to act as acceptance tests. If there is a test failure, an exception may be raised. Alternatively, it may be sufficient to return an error indication to the calling unit. Post-conditions can be used to specify expected results when carrying out unit testing [18].

Many requirements must be considered in the design of a system that supplied with the embedded feature of exception handling. Such as the potential actions stimulating the exclusions, the influences of those events on the system, and the choice of fitting recovery actions. For a software module, there are three classes of exception that are activating events as follows:

- Interface Exceptions. These exceptions are activated by the self-protection tools of a module when it detects an unacceptable service request. Then these exceptions will be controlled by the module that requested the service.

- Local Exceptions. These exceptions are activated by the error-detection mechanisms of a module when it detects an error in its in-house processes. Then these exceptions will be controlled by the module’s fault tolerant abilities.

- Failure exceptions. These exceptions are activated by a module when it detects an error, but its fault processing mechanism could not correctly handle this error. So, failure exceptions inform the module requesting the service that several other means should achieve its function.

The concept of error containment and isolation is fundamental and must be considered in the design of a system that embeds and support exception handling features because this will lead to the design of capable exception handlers. These exception handlers must be supported with a proper design of system structure, actions, and error detection mechanisms to enclose and isolate the effects of errors within a specific group of interrelating components at what time the error is sensed.

There is an argument that there is a difference between fault tolerance and exception handling. “The difference between fault tolerances versus exception handling is that exception handling deviates from the specification and fault tolerance attempts to provide services compliant with the specification after detecting a fault. This is an important difference to realize between trying to construct robust software versus trying to construct reliable software.
Reliable software will accomplish its task under adverse conditions while robust software will be able to indicate a failure correctly, (hopefully without the entire system failing)" [11].

A1.4 Checkpoint and Restart: The checkpoint and restart is the most commonly used mechanism among a few other recovery mechanisms that applied for single-version software (see Figure. 5). This mechanism is usually used to recover a particular type of software faults (in view of that majority of the software faults remaining after development are unforeseen, state-dependent faults).

This kind of fault behaves in the same way that the transient hardware faults are behaving: they come out, do the damage, and then disappear, leaving behind no apparent reason for their activation in the first place. Hence, the checkpoint and restart mechanism handle this type of fault according to its characteristics by merely restarting a module. This is usually sufficient to permit the module to accomplish its execution. A restart, or backward error recovery, has many advantages such as it is autonomous of the damage produced by a fault, valid to unexpected faults, general enough that it can be used at numerous levels in a system, and conceptually simple.

There are dual types of restart recovery mechanisms:

1) Static Restart Recovery mechanism. This mechanism considers the operational condition at the moment that the error is detected, to return the module execution state straightforwardly to the preliminary reset state or one of a set of potential states. All these states are predefined and predetermined states.

2) Dynamic Restart Recovery mechanism. This mechanism returns the module execution state after an error is detected, to one of a set of "dynamically created checkpoints that are snapshots of the state at different points during the execution. These checkpoints are either created at fixed intervals or particular points during the system operation, where some optimizing rule determines these points. These checkpoints have the main advantage which is that they depend on states created during the system operation, so this will permit to implement forward progress of system execution without having to throw away all the process and computation done until the time of error detection".

There is an important issue that must be considered when there is an intention to use a restart recovery mechanism, which is the existence of unrecoverable cases. These cases are related to systems that have external activities that cannot be managed and handled by a simple process of returning the system execution to one of the predefined static points (states) or one of the dynamically created checkpoints (states). "Examples of unrecoverable cases include firing a missile or soldering a pair of wires. These actions must be given special treatment, including compensating for their consequences (e.g., undoing a solder) or just delaying their output until after additional confirmation checks are complete (e.g., do a friend-or-foe confirmation before firing)".

A1.5 Process Pairs: This technique uses dual same software varieties and makes them running on separate dual processors (see Figure. 6). Also, this technique implements checkpoint and restart as a recovery mechanism. In this technique, one of the dual processors indicated as the primary processor and the other as secondary processor. When the system starts running, the primary processor will be the one who is responsible for executing the system operations. So, the primary processor will process the input and produces the output while creating checkpoint information that is sent to the secondary processor. Thus, the secondary processor will act as a backup processor.

As an error is sensed, the secondary processor loads the last checkpoint as its initial state and takes over the task of the primary processor. In other words, the secondary processor will become the primary processor. Then, the faulty processor (the previous prime processor) becomes offline and initiates diagnostic checks. And maybe, maintenance and replacement are implemented on the defective processor.

When the faulty processor is repaired and became ready to operate again, it returning to service and becomes the secondary processor and begins receiving checkpoints information from the primary processor.

The continuous delivery of services is the most important advantage of this recovery technique, where the system does not interrupt its operation even after the occurrence of a failure.

A1.6 Data Diversity: This technique applies "input sequence workarounds" to recover system faults. It is usually combined with checkpoint and restart recovery technique. Data diversity considered as the last line of defense in contradiction of system design faults.
“This technique is based on the concept that faults in deployed software are usually input sequence dependent. Data diversity may improve the performance of the checkpoint and restart recovery technique by employing different input re-expressions on each retry (see Figure 7). Each retry action aims to produce output results that are either precisely equivalent or semantically equivalent in some way. In general, the notion of equivalence is application dependent”.

Data diversity can be classified into three models:
- Input Data Re-Expression, where just the input is reformed (see Figures 7 and 8).
- Input Re-Expression with Post-Execution Adjustment, where the output is also processed as necessary (with addition to input changing) to get the required output value or format (see Figure 9).
- Re-Expression via Decomposition and Recombination, “where the input is divided into smaller elements and then recombined after processing to produce the required output” (see Figure 10).
“Also, Data diversity can be combined with Process Pairs technique through employing different re-expressions of the input in the primary and secondary processors. It is rational to be able to have as a feature some degree of execution flexibility into the design of the software components modules) to make the application of data diversity concept easier”.

“Finally, data diversity could be used in conjunction with the multi-version fault tolerance techniques. The components, produced through the design diversity, are called variants (originally called alternates in the recovery block scheme and versions in the N-version programming scheme)” [26].

A2 Multi-Version Software Fault Tolerance Techniques:

"Multi-version software fault tolerance techniques are based on the assumption that: components built differently (i.e., different designers, different algorithms, different programming languages, etc.) should fail differently". Therefore, if a failure occurred in one of the versions on a specific input, there is at least one of the other varieties should have the ability to provide accurate or acceptable output. So, depending on the previous assumption, multi-version fault tolerance uses dual or more versions (or “variants”) of a program (software). These versions are running either in sequence or in parallel. There are dual cases about how versions are used as alternatives (with a separate method of error detection), they either be in pairs (to apply detection by replication checks) or be in larger groups (to use masking via voting).

We can classify these techniques into dual groups as follows: Traditional and New Techniques.

A2.1 Recovery Blocks: The Recovery Blocks technique is one of the conventional and earliest developed techniques for multi-version software fault tolerance (see Figure 11). This technique is based on combining the basic concepts of a checkpoint and restart mechanism with multiple versions of software components (modules). The main issue in this technique is to use different developing methods and approaches (i.e., different algorithms, various programming languages, etc) in building the multiple versions, so as to try to ensure that at least there is one of the versions (alternate modules) be able to tolerate the component (module) or the system failure.

Thus, the recovery blocks technique consists of a Primary Version (main block or modules) and at least one Alternate Version (secondary block or module), where the execution of these manifold varieties can be sequential or in parallel depending on the existing processing capability and system requirements. Also, there is another basic unit in this technique, which is the Acceptance Test (adjudicator) component that exists on exit from a primary or alternate block. Where this component is responsible for testing the version (primary or alternate) output for its validity. Supposing that the programmer can build a satisfactorily uncomplicated adjudicator, will produce a system which is problematic to enter into a fault state.

The operational mechanism of the recovery block technique is as follows: in the beginning, the system will adopt the execution of the primary version to perform its computations. When the acceptance test indicates that this version fails as a result of detecting faulty output from it, the system will switch to one of the alternate versions to continue its computations, starting from a valid operational point (state). “This point is determined by checkpoint, where checkpoints are created before a version executes 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 most of the time successfully, 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. 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” [2, 26].

One of the most challenging (and critical) aspects of the recovery blocks is the design of the acceptance tester. Three different methods may be used [18]:

- Test result against pre-defined values (e.g., checking that values lie within valid ranges).
- Test result against predicted values. In dynamical systems, for example, the maximum rates of change of parameters can be used for this purpose. Employing this we can predict the maximum possible change of parameter values in any time interval. The actual value produced by the algorithm should not exceed the predicted amount.
- Employing the output value, compute the input values which should have produced this output. Compare these with the checkpoint values to see if they agree. This technique (an inverse or ‘reverse’ algorithm check) can be applied in general to control and signal processing algorithms where time isn’t a problem.

Note that we must know system or software attributes to form acceptance tests.
A2.2 N-Version Programming: N-Version programming technique is one of the most common procedures that are used in multi-version software fault tolerance techniques (see Figure 12). In this technique, the multiple versions are built by different design and development teams depending on standard requirements and specifications. These versions are typically executed in parallel on separate processors. Also, they can be run on single processor systems in efficient way.

In addition to the multiple versions, N-versions technique also includes a selection algorithm (usually called voter) component. The voter receives the results (outputs) of the multiple versions as input and compares these entire results one with each other to decide the selection of the correct result (output). This decision is made by voting on the comparison and depending on the comparison result an appropriate action will be taken. The comparison result may be one of the following cases:

If all N-Versions results (outputs) are identical, then the system will send the result (output) to the next stage of processing. Otherwise;
- If not all but there is a majority of the results (outputs) are identical then the system will send the result (output) of this majority to the next stage of processing (typically there are N – 1 agreements at any one time). Otherwise;
- If there is no majority agreement, then the component (module) should raise an exception to inform rest of the system its failure to finish its task.

This technique can tolerate software bugs that affect marginal versions, but cannot tolerate correlated fault (the reason for failure is common to dual or more modules) [5].

"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" [2].

Design of the voter can be sophisticated by the need to use techniques to perform inexact voting [26].

Many researches have been concerned on development of methodologies that raise the likelihood of realizing active diversity in the software versions.

Research has demonstrated that the arguments for reliability through diversity (N-version programming) are not always valid. When developing software from the same specification, different teams made the same mistakes. Software redundancy did not give the theoretically predicted increase in system reliability. Furthermore, if the specification is incorrect, all versions will include the common specification errors. This does not mean that N-version programming is useless. It may lessen the absolute number of failures in the system. N-version programming gives increased confidence but not absolute confidence in the system reliability [2].

There are not many but important differences between the N-versions technique and recovery block technique, as follows:
- The main difference between these dual techniques is that the N-versions technique usually uses one generic decision algorithm (voter or decider), while the recovery blocks technique uses an acceptance tester (adjudicator) for each module which is application dependent.
- The other important difference is that, in the beginning, and also the conventional case, the multiple versions in the recovery blocks are executed on sequential manner. Later, the recovery block technique has been extended to include concurrent execution of the various alternatives. The N-versions programming is always designed to execute the multiple versions in parallel on separate processors.
- So, in a sequential retry system, the cost in time of trying multiple alternatives may be very expensive, especially for real-time system applications. Conversely, concurrent systems need the expense of N-way hardware and communications systems to connect them.

Both techniques have the advantages and disadvantages of the engineering interchanges, especially economic costs, involved with developing. It is a critical issue for the engineer to consider these costs when deciding the best technique to be implemented in his project.
A2.3 N Self-Checking Programming: This technique is based on employing manifold software varieties joint with modified models of the Recovery Blocks and N-Version Programming. Hence, dual models of this technique can be developed. The first one is the N Self-Checking programming employing acceptance tests (see Figure. 13). The implementation of this model based on applying some modifications on recovery blocks technique, where there will be separate acceptance tests for each version (where the development of the versions and the acceptance tests are made self-reliant from conjoint necessities). This is the essential difference between this N self-checking model and the recovery blocks technique. Also, each version will be attached with a rank which its degree depends on the extent of reliability that expected for each version. However there is a similarity between these dual techniques, which is the both techniques can execute their versions (and their acceptance tests in the case of the N self-checking model) either in sequence (there is a need for employing checkpoints in this case) or in parallel (there is a need for employing input and state consistency algorithms in this case) but the output from the highest-ranking version that passes its acceptance test successfully will be adopted.

The second model of the N Self-Checking programming is N Self-Checking programming employing comparison for each pair of varieties to detect errors (see Figure. 14). The implementation of this model based on employing a decision algorithm to select the correct output. The advantage of this model is that the development of the decision algorithm is easier than the development of the acceptance test because the decision algorithm is application independent. This self-checking programming model has the theoretic fault of encountering conditions where numerous pairs pass their comparisons every with dissimilar outputs. This situation must be considered, and a proper decision procedure should be selected during design.

A2.4 Consensus Recovery Blocks: This technique is based on combining N-Version Programming technique and Recovery Blocks technique to gain better reliability than that achieved by employing each technique alone (see Figure 15). In other words, the Consensus Recovery Blocks technique is an integration of N-Version Programming technique and Recovery Blocks technique to try to overcome the shortages and difficulties those are embedded in each of those techniques when they designed and used as individuals.

The main difficulty that may encounter the designers is building proper acceptance tests because they are applications dependent and there is no standardization for their development so that these reasons may lead to the occurrence of design faults.

The main lack in N-Version Programming technique is that the use of voters as a decision mechanism cannot be applied in all cases because, in some applications systems, there is a possibility of multiple correct outputs. In this case, the voter will fail in selecting the appropriate output.

So, the Consensus Recovery Blocks technique develops a validity component that consists of dual layers. The first layer uses an output selection algorithm (voter) similar to that used in N-Version Programming technique. If this layer cannot select appropriate output, it will declare a failure, and the system will switch into and activate the second layer where this layer uses an output selection algorithm (acceptance test) similar to that employed in Recovery Blocks technique.

There is a critical issue attached to the developing process of Consensus Recovery Blocks technique. Theoretically, according to the reliability concepts, this combined approach has the likelihood of producing a more steadfast piece of software, but it will be much more complex than either of the individual techniques. Hence, there should be a consideration that the added complexity could work in contradiction of the system design in such a way that makes the design less reliable.
A2.5 $t/(n-1)$-Variant Programming [2]: “The foremost difference amid this approach and the ones mentioned above is in the used mechanism to choose the output from manifold variants. The design of the selection logic is by the theory of system-level fault diagnosis. Fundamentally, a $t/(n-1)$-VP architecture consists of $n$ variants and uses the $t/(n-1)$ diagnosability measure to isolate the faulty units to a subset of size at most $(n-1)$ assuming there are at most $t$ faulty units. Thus, at least one non-faulty unit exists such that its output is correct and can be used as the result of computation for the module”.

"$t/(n-1)$-VP compares favorably with other approaches in that the complexity of the selection mechanism grows with order $O(n)$ and it can potentially tolerate multiple dependent faults among the versions. It also has a lower probability of failure than N Self-Checking Programming and N-Version Programming when they adopt an uncomplicated voter as selection logic”.

The following table (table 1) illustrates the extended metrics characteristics for the above multi-version software fault tolerance techniques [27]:
B. New Software Fault Tolerance Techniques:

SFT have new techniques such as adaptive n-version systems, fuzzy voting, abstraction, rejuvenation, and parallel graph reduction. [28]

B1. Adaptive n-version Systems: "The n-version approach has the following tradeoffs: A software/hardware error in a version might not only cause a deviation at the time-point of its emergence but might additionally corrupt data needed for subsequent (and error-free) parts of the associated process. So, the question arises, how far a software/hardware component once deviating from the majority behavior, can later be relied on". [28, 29]

"An adaptive approach is used for considering the different versions of the system. Here, adaptivity mainly means that the rigid organization of static redundancy methods is changed. It is an adaptive approach which allows to model and manages different quality levels of the system versions. This is achieved by introducing an individual weight factor to each version of the n-version system. This weight factor is then included in the voting procedure -the voting as based on a weighted counting-. Additionally, the extension of this approach to component-based strategies for building the individual versions of the n-version system as considered". [29, 30, 31]

"At the voting stage, the voting is based on a weighted counting of the number of monitored events for the deviation behavior of the individual version. The voting procedure can be adaptively modified and tailored to the fault state of the overall system". [28]

B2 Fuzzy Voting: "Voting algorithms are used to arbitrate between the results of redundant modules in fault-tolerant systems. The inexact majority and weighted average voters have been used in many applications, although both have problems associated with them. Inexact majority voters require an application-specific 'voter threshold' value to be specified, whereas weighted average voters are unable to produce a benign output when no agreement exists between the voter inputs. Neither voter type can cope with uncertainties associated with the voter inputs". [32]

"It softens the harsh behavior of the inexact majority voter in the neighborhood of the ‘voter threshold’, and handles uncertainty and some multiple error cases in the region defined by the fuzzy input variables. The voter assigns a fuzzy difference value to each pair of voter inputs based on their numerical distance. A set of fuzzy rules then determines a single fuzzy agreeability value for each data which describes how well it matches the other inputs. The agreeability of each voter input is then DE fuzzified to give a weighting value for that input which determines its contribution to the voter output. The weight values are then used in the weighted average algorithm for calculating the final voter output. The voter is experimentally evaluated from the point of view safety and availability, and compared with the inexact majority voter in a Triple Modular Redundant structured framework. The impact of changing
some fuzzy variables on the performance of the voter is also investigated. Fuzzy relation is the degree of interconnecting elements of sets that comprise the relation. A fuzzy relation is a fuzzy equivalence relation if and only if all three properties of fuzzy relations are satisfied: reflexivity, symmetry, and transitivity. Fuzzy equivalence relation results in more reliable systems". [28, 32]

"Fuzzy logic aims at modeling the imprecise modes of reasoning emulating human approximation approach to make a rational decision in an environment of uncertainty and imprecision. The fuzzy approach is an alternative design methodology enabling solutions to complex problems employing probabilistic logic and probability theory". [33]

**B3 Abstraction:** "Abstraction is a method of the SFT used to reduce costs. It improves availability because each replica can be repaired periodically employing an abstract view of the state stored by correct replicas and because each replica can run distinct or non-deterministic service implementations, which reduces the probability of common mode failures. The replicated file system required only a modest amount of new code, and preliminary performance results indicate that it performs comparably to the off-the-shelf implementations that it wraps. Software fault tolerance employing replication is expensive to deploy. Abstraction will Abstract Specification Encapsulation, which uses abstraction to reduce the cost of fault tolerance and to improve its ability to mask software errors. Abstraction hides implementation details to enable the reuse of off-the-shelf implementations of important services (e.g., file systems, databases, or HTTP daemons) and to improve the ability to mask software errors". [29, 34]

**B4 Rejuvenation:** "Software rejuvenation is the concept of gracefully in terminating an application and immediately restarting it at a clean internal state. When software applications execute continuously for long periods (scientific and analytical applications run for days or weeks, servers in client-server systems are expected to run forever), the processes corresponding to the software in execution age or slowly degrade concerning the effective use of their system resources. The causes of process aging are memory leaking, unreleased file locks, file descriptor leaking, data corruption in the operating environment of system resources, etc. Process aging will affect the performance of the application and eventually cause the application to fail". [35]

"Software rejuvenation is a periodic preemptive rollback of continuously running applications to prevent failures in the future". "Rejuvenation follow the policy of Proactive fault tolerance that is based on predicts the faults proactively and replace the suspected components by other working components thus avoiding recovery from faults and errors. In Rejuvenation, the system is planned for periodic reboots, and every time the system starts with a new state". [29, 36]

**B5 Parallel Graph Reduction:** "Recently, parallel computing is popularly applied to many systems. Functional programming is suitable for parallel programming because of its referential transparency and independence among each program. Referential transparency means that all references to the value are therefore equivalent to the value itself and the fact that the expression may be referred to from other parts of the program is of no concern. It is applied to symbol processing systems and parallel database systems. Programs of some functional programming can be regarded as graphs and are processed in terms of reduction of the corresponding graphs. A fault tolerance scheme based on parallel graph reduction in functional programming was proposed. The method is a class of receiver-based message logging and time overhead of fault tolerance is reduced by taking advantage of referential transparency". [29]

**VI. CONCLUSIONS**

In this study, we presented a review of the most common software fault tolerance traditional techniques and some of the most promising new methods.

We noted that the failure of software systems is caused by design faults, for the reason that we are not able to produce completely fault-free software yet due to the high complexity of present-day software systems. Also, software faults have a tendency to be state-dependent and initiated by particular input sequences which cannot be predictable most of the times. Even though component reliability stands for an essential quality measure for system level analysis, software reliability is difficult to estimate, and the procedure of post-verification reliability estimates is still a controversial issue. In several applications, software safety is more important than reliability, especially in safety-critical systems when there is no safe state and the safety of the system depends on its availability. So, software fault tolerance techniques should be used in those applications to prevent catastrophes. Besides, software fault tolerance techniques are implemented with the software systems to make the outputs of these systems more reliable.

Most of SFT techniques are designed and structured based on the following main concepts:

- Process pairs utilize the state dependence feature of the majority of software faults to permit continuous delivery of services regardless of the activation of faults.
- Data diversity targets are for averting the activation of design faults through trying manifold alternative input sequences.
- Multi-version techniques are dependent on the notion that software built in a different way must fail in a different way and thus if one of the redundant versions
Advantages they also include critical disadvantages, such as:

- In recovery blocks technique: the complexity of designing appropriate acceptance tests, and late results.
- In N-version programming technique: contemporary systems need the expenditure for N-way hardware and communications to connect them, and there is a possibility that all the versions have different outputs.
- In N self-checking programming technique, if one of the versions produces a result, which is only slightly different from the other, the acceptance test may not be able to determine that it is incorrect.
- In consensus recovery blocks technique: output cannot be produced when the computed results differ but there is more than one result that satisfies the acceptance tests.

REFERENCES


DOI 10.5013/IJSSST.a.20.03.07
ISSN: 1473-804x online, 1473-8031 print