Fault Tolerant Engineering in a Conceptual Design

Saman Aliari Zonouz, Jafar Habibi
aliari@ce.sharif.edu, habibi@sharif.edu
Department of Computer Engineering
Sharif University of Technology

Abstract

Many risks usually plague a software development life cycle (SDLC). The relationship between product quality and process capability and maturity has been recognized as a major issue in software engineering based on the premise that improvements in process will lead to higher quality products. This paper presents fault tolerant software engineering (FTSE) in a conceptual design. The concepts of FTSE are based on three types of redundancy: the human resource redundancy, the time redundancy, and the software redundancy. Employing fault tolerance techniques from the beginning steps of SDLC makes it possible to choose the appropriate models, design and infrastructure in order to develop reliable software. Moreover, it results in clearer program code, increased readability, less maintenance overhead, and delivers adequate performance.

Key words: *Fault tolerance, Software Engineering, Fault tolerant engineering, Software development life cycle.*

1. Introduction

Generally, Software reliability engineering (SRE) stems from the needs of software users [1]. Nowadays operations are increasingly more dependent on software-based systems and tolerance of failures of such systems is decreasing because of their growing complexity. Software engineering is not only expected to help deliver a software product of required functionality on time and within cost; it is also expected to help satisfy certain quality criteria [2]. The most prominent one is reliability. SRE is the applying science of predicting, measuring, and managing the reliability of software-based systems to maximize customer satisfaction [1, 2, 3, 4].

Many risks usually plague a software development life cycle (SDLC). Risk analysis and management are a series of steps that help a software team to understand and manage such risks [5, 6, 7, 8, 9, 10, 11, 12, 13]. Risk management steps include risk identification, risk analysis, risk ranking, and planning to manage highly probable risks [5].

In addition to risk management, measuring and evaluating the stability of engineering process is important because of the recognized relationship between process quality and product quality [14]. Stability is the condition of a process that results in increasing reliability, decreasing risk of deployment, and increasing test effectiveness. Moreover, our focus is on process stability, not code stability.

We concentrate on the chief quality factor, i.e. reliability. According to Lehman, large projects are never completed; they just continue to evolve [15]. In other words, with software, we are dealing with a moving target; therefore, a reliable Software Development Life Cycle (SDLC) is required.

Khoshgoftarr et al used discriminated analysis in each of iterations of their project to predict fault prone modules in the next iteration [16]. This approach provided an advance indication of reliability and the risk of implementing the next iteration. This study deals with product reliability but does not address the issue of process stability.

The objective of this paper is to employ fault tolerance concepts throughout software development process. Having fault tolerance in mind from the beginning allows software developers to engineer support for fault tolerance which makes it possible to state the required level of fault tolerance precisely and then choose the appropriate models, design and infrastructure to achieve it. Moreover, it helps reduce the complexity of fault tolerance which results in clearer program code, increased readability, less maintenance overhead, and delivers adequate performance. For this purpose, error confinement techniques provided by well-known fault tolerance models, error recovery, and structured error location will be used; consequently, the resulting design will make use of fault tolerant infrastructures.

This paper is organized as follows: in section 2, related work will be described. Part 3 is devoted to a brief clarification of fault tolerance concepts. In parts 4, 5, and 6, human resource redundancy, time redundancy, and software redundancy will be explained in order.

2. Related Work

A number of useful related process and maintenance measurement projects have been reported in the literature. Briand, et al, developed a process to characterize software maintenance projects [17]. They present a qualitative and inductive methodology for performing objective project characterizations to identify maintenance problems and needs. This methodology aids in determining causal links

between maintenance problems and flaws in the maintenance organization and process. Although the authors' have related ineffective maintenance practices to organizational and process problems, they have not made a linkage to product reliability and process stability.

Frankl et al has developed an approach that is to provide mechanisms to improve reliability of software after it has been implemented [18]. They use testing techniques to identify faults in the software that are likely to cause failures. Although they carry out an important research agenda, we believe that is cheaper to design and evaluate dependability concerns in the early stages of software engineering process.

Henry, et al, found a strong correlation between errors corrected per module and the impact of the software upgrade [19]. This information can be used to rank modules by their upgrade impact during code inspection in order to find and correct these errors before the software enters the expensive test phase. The authors treat the impact of change but do not relate this impact to process stability.

Khoshgoftarr et al used separate analysis in each of iterations of their project to predict fault prone modules in the next iteration [16]. This approach provided an advance indication of reliability and the risk of implementing the next iteration. This study deals with product reliability but does not address the issue of process stability.

Pearse and Oman applied a maintenance metrics index to measure the maintainability of C source code before and after maintenance activities [20]. This technique allowed the project engineers to track the "health" of the code as it was being maintained. Maintainability is assessed but not in terms of process stability.

Pigoski and Nelson collected and analyzed metrics on size, trouble reports, change proposals, staffing, and trouble report and change proposal completion times [21]. A major benefit of this project was the use of trends to identify the relationship between the productivity of the maintenance organization and staffing levels. Although productivity was addressed, product reliability and process stability were not considered.

Sneed reengineered a client maintenance process to conform to the ANSI/IEEE Standard 1291, Standard for Software Maintenance [22]. This project is a good example of how a standard can provide a basic framework for a process and can be tailored to the characteristics of the project environment. Although applying a standard is an appropriate element of a good process, product reliability and process stability were not addressed.

Stark collected and analyzed metrics in the categories of customer satisfaction, cost, and schedule with the objective of focusing management's attention on improvement areas and tracking improvements over time [23]. This approach aided management in deciding whether to include changes in the current release, with possible schedule slippage, or include the changes in the next release. However, the authors did not relate these metrics to process stability.

Schneidewind has integrated product and process of software development to propose a unified product and process measurement model for product evaluation and process stability analysis [24]. He concluded, based on both predictive and retrospective use of reliability, risk, and test metrics, that it is feasible to measure and assess both product quality and the stability of a maintenance process.

3. Fault Tolerant Engineering

A successful software project requires a reliable process which guarantees to accomplish the project. As mentioned above, conventional approaches cause to minimization or elimination of the predictable or unpredictable risks. Considering the development process itself as a system composed of components (i.e. phases of the SDLC), helps us to realize the existing approaches aid to increase the reliability of the process without employing fault tolerance concepts, instead they attempt to augment the reliability of the whole process by replacing the current components with the more reliable ones.

We propose a fault tolerant process in a conceptual design that will help to have a reliable SDLC. It is important to declare that we focus on the process model, not on the product (e.g. software system). The offered process includes fault detection, fault containment, fault location, fault recovery, and fault masking.

A system which employs fault masking achieves fault tolerance by hiding faults that occur. Such a system needs fault containment rather than fault detection; in other words, we localize the effects of a fault. Systems which do not employ fault masking require fault detection, fault location, and fault recovery to achieve fault tolerance. Fault tolerance of a system is usually performed using some form of redundancy [25]. We consider redundancy as categorized into three groups: human resource redundancy, time redundancy, software redundancy.

4. Human Resource Redundancy

We propose three basic forms of human resource redundancy: passive, active, and hybrid. Passive techniques use the concept of fault masking to hide the occurrence of faults in the process and prevent the faults resulting in project failures. Passive approaches are designed to achieve fault tolerance without requiring any action on the SDLC phases.

The active approach (i.e. the dynamic method), achieves fault tolerance by detecting the existence of faults and performing some action to remove the faulty human resource or team from the development process. In other words, active techniques require that the process be reconfigured to tolerate faults; therefore, active approach employs fault detection.

Hybrid techniques combine the attractive features of both the passive and active approaches. Fault masking is used in hybrid processes to avoid failures. Fault detection, fault location, and fault recovery are also used in the hybrid approaches to improve fault tolerance by removing faulty team and replacing it with spares.

4.1. Passive Resource Redundancy

Passive human resource redundancy relies on voting mechanisms to mask the occurrence of faults in working teams. Most passive approaches are developed around the concept of majority voting. Triple modular redundancy, as the most important passive redundancy technique, is clarified.

4.1.1. Triple Modular Redundancy

The most common form of passive human resource redundancy is called triple modular redundancy (TMR). The basic concept of the TMR is to triplicate the working team on a same issue and perform a majority vote to determine the output of the development phase which requires employing someone as the voter. If one of the modules (i.e. working teams) becomes faulty, the two remaining fault free modules mask the results of the faulty module when the majority vote is performed. A sample TMR of analysis and design team has been illustrated in Figure.1.

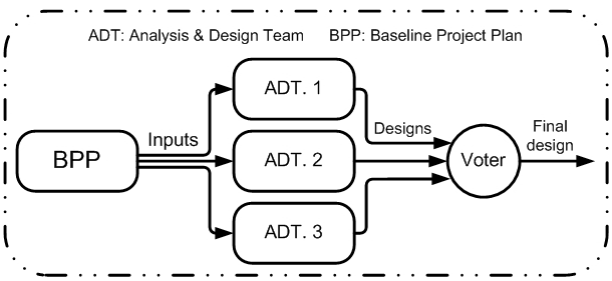

Figure 1: A sample TMR of analysis and design team

The primary difficulty with TMR is the voter; if the voter fails (e.g. a bad decision), the complete phase fails; therefore voter is usually considered as a single point of failure. Several techniques can be used to overcome the effects of voter failure. One approach is to triplicate the voters and provide three independent outputs.

The three teams each receives identical inputs and perform identical functions using those inputs. The results generated by the teams are voted on to produce three results. Each result is correct as long as no more than one module, or input, is faulty. One method to vote is to have a parameterized mathematical expression in order to select one of the outcomes.

A generalization of the TMR approach is the N-modular redundancy (NMR) technique. NMR applies the same principle as TMR but employs N teams as opposed to only three. In most cases, N is selected as an odd number so that a majority voting arrangement can be used.

4.2. Active Resource Redundancy

Active human resource redundancy techniques attempt to achieve fault tolerance by fault detection, fault location, and fault recovery. In other words, this approach does not attempt to prevent faults from producing failures within the development process.

4.2.1. Duplication with Comparison

Duplication with comparison is an example of active human resource redundancy. The basic concept of duplication with comparison is to employ two technical teams, have them perform the same jobs in parallel, and compare the results of their function which requires employing someone as the comparator. In the event of a disagreement, an error message is reported to the manager; therefore, a meeting is required to decide which one to select.

4.2.2. Standby Sparing

Second form of active human resource redundancy is called the Standby sparing (or standby replacement) technique in which, one of the technical teams is operational and one or more teams serve as standbys, or spares. If a fault is detected and located, the faulty team is removed from development process by the management and replaced with a spare team (See Figure.2).

Standby sparing can bring a development process back to full operation capability after the occurrence of a fault, but it requires that a momentary disruption in performance occur while the reconfiguration, i.e. replacing the faulty team with the spare one, is performed. If the disruption in process must be minimized, hot standby sparing can be used.

In the hot standby sparing technique, the spare teams observe working team and know as much as them; therefore, spare teams are prepared to take over at any time. In contrast to hot standby sparing is cold standby sparing where the spare teams do not have information about the working team, their inputs, or outputs until needed to replace a faulty team.

The disadvantage of cold standby sparing approach is the time required to train the spare team and perform initialization prior to bringing the team into active service. The advantage of cold standby sparing is that spare teams do not use resources until needed to replace a faulty team.

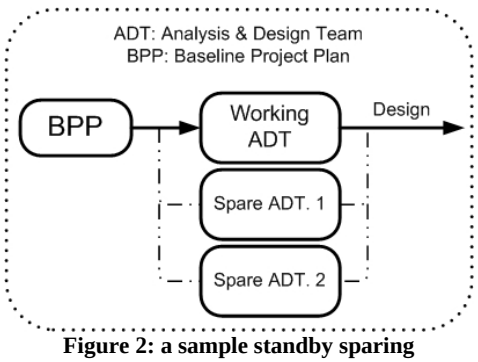

Figure 2: a sample standby sparing

4.2.3. Pair and Spare

The pair and a spare technique, combines the features present in both standby sparing and duplication with comparison. In essence, the Pair and a spare approach uses standby sparing; however, two teams are working in parallel at all times and their results are compared to provide the error detection capability required in the standby sparing approach. As soon as a fault is detected, the management initiates the reconfiguration process that removes faulty team and replaces it with a spare one.

When the faulty team cannot be distinguished, a variation on the pair and a spare technique can be used that is to always employ teams in pairs. During the development process, teams are permanently paired together, and when one team fails or become faulty, neither team in the pair is used.

4.2.4. Hard Deadlines

One form of human resource redundancy that is extremely useful for detecting faults in a development process is the hard deadline based approach. A hard deadline management is an active form of human resource redundancy; because some action is required on a phase of a SDLC to indicate a fault-free status.

The basic concept of the hard deadline management is that the lack of an outcome on a milestone indicates existence of a fault. A hard deadline manager is a supervisor who starts monitoring of a phase as soon as the process starts. The failure of the process to perform the hard deadline manager resets or cancels the process to prevent a SDLC failure from occurring.

4.3. Hybrid Resource Redundancy

The fundamental concept of hybrid human resource redundancy is to combine the attractive features of both the active and the passive approaches. Fault masking is employed to prevent the working teams from producing erroneous results; and fault detection, fault location, and fault recovery are used to reconfigure the development process in the event of a fault. Hybrid redundancy can be very expensive in terms of the amount of human resource required to maintain the development process.

4.3.1. N-modular Redundancy with Spares

The idea of N-modular redundancy (NMR) with spares is to provide N modules (i.e. technical teams) arranged in a voting, or a form of voting, configuration. In addition, spare teams are provided to replace faulty ones.

The benefit of NMR with spares is that a voting configuration can be restored after a fault has occurred. For example, a development process that uses TMR with one spare will mask the first team fault that occurs. If the faulty team is then replaced with the spare one, the second team fault can also be masked, thus providing tolerance of two team faults. For a passive approach to tolerate two team faults, five teams must be configured in a fault masking arrangement. The hybrid approach can accomplish the same results using only four teams and some fault detection, location, and recovery techniques.

4.3.2. Self-Purging Redundancy

A second approach to hybrid redundancy is called self-purging redundancy [26]. The basic concept of self-purging redundancy is similar to that of the NMR with spares approach.

The major difference is that all teams are actively participating in the development process in the self-purging technique, whereas some teams function as spares in the NMR approach and may not be an active part of the development process until a fault occurs. Each of the N teams is authorized to remove itself from the process in the event that its output disagrees with the voted output of the process.

There are two basic features of the self-purging redundancy concept. Firstly, N teams are obtained. Each team is capable of performing the functions required of the development process. Second, a voter is employed to produce the development process result and provide masking of any faults that occur.

4.3.2. Sift-Out Modular Redundancy

Another hybrid redundancy method is called sift-out modular redundancy [27]. As illustrated in Figure.3, Sift-out modular redundancy also uses N identical teams that are configured into a development process employing specialists called comparators, detectors, and collectors.

The responsibility of the comparator is to compare each team's output with the remaining teams' outputs. Thus, the comparator compares every two outputs with each other and reports each comparison that is performed.

The role of the detector is to determine which disagreements are reported by the comparator and to remove a team that disagrees with a majority of the remaining teams.

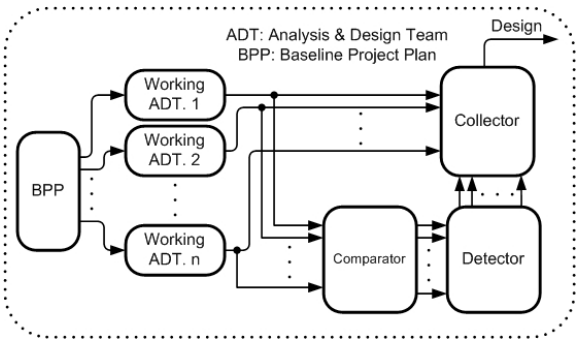

Figure 3: Sift-out Modular Redundancy

The last major individual of the sift-out modular redundancy approach is the collector. The responsibility of the collector is to report the system's output, given the outputs of the individual teams and the reports from the detector that indicate which teams are faulty. A module that is properly identified as faulty is not allowed to influence the output of the system.

4.3.3. Triple-Duplex Architecture

The final hybrid redundancy technique is called the triple-duplex architecture because it combines duplication with comparison and triple modular redundancy. The use of TMR allows faults to be masked and continuous, error-free performance to be provided for up to one faulty team. The use of duplication with comparison allows faults to be detected and faulty teams removed from the TMR voting process.

5. Time Redundancy

The fundamental problem with the forms of redundancy discussed thus far is the penalty paid in extra human resource for the carrying out the various techniques. Human resource redundancy can require large amounts of extra human resource for their implementation. In an effort to decrease the human resource required to achieve fault detection or fault tolerance, time redundancy is considered as follows.

Time redundancy methods attempt to reduce the amount of extra human resource at the expense of using additional time. In many SDLCs, the time is of much less importance than the human resource because human resource is a physical entity that impacts total cost, staff size and responsibilities. Time on the other hand, may be readily available in some development processes.

The selection of particular type of redundancy is very dependent upon the SDLC. For example, some projects can better stand additional human resources than additional time; others can tolerate additional time much more easily than additional human resources. The selection in each case must be made by examining the requirements of the process and the available techniques that can meet such requirements.

5.1. Transient Fault Detection

The basic concept of time redundancy is the repetition of a phase in ways that allow faults to be detected. The most basic form of time redundancy is to accomplish the same process two or more times and compare the results if a discrepancy exists. If an error is detected, the process can be performed again to see if the disagreement remains or disappears. Such approached are often good for detecting errors resulting from transient faults (e.g. programming faults), but they cannot protect against errors resulting from permanent faults.

Time redundancy can often be employed to distinguish between the permanent and the transient faults. The processes can be performed one or more times after the detection of the first error; if the error condition clears, the fault that caused the error can be assumed to have been transient. If, however, the problem continues to be detected, the fault is most likely permanent, and the faulty teams of the SDLC must be removed.

5.2. Permanent Fault Detection

One of the biggest potentials of time redundancy, however, appears to be the ability to detect permanent faults while using a minimum of extra human resources. REDWC [28] is considered as follows.

5.2.1 Re-performing with Duplication with Comparison

An alternative method that takes advantage of both time redundancy and human resource redundancy concepts is called re-performing with duplication with comparison (REDWC). The method with which error detection is accomplished resembles that of duplication with comparison. Time redundancy technique is then employed to complete the development process and obtain the final result.

REDWC is similar in many respects to a method described in [29]. Re-performing for error detection can be accomplished using time redundancy techniques; therefore, The time redundancy approach can provide for error correction if the computations are repeated three or more times.

6. Software Redundancy

In software development life cycle (SDLC), many fault detection and fault tolerance techniques can be implemented in software. The redundant human resource necessary to implement the capabilities can be minimal, whereas the redundant software can be substantial. Redundant software can occur in many forms; you do not have to replicate complete programs to have redundant software. Software redundancy can appear as several extra lines of code used to check the result of a particular development process.

6.1. Consistency Checks

A consistency check uses a prior knowledge about the characteristics of the result of a specific SDLC phase to verify the correctness of the result. For example in some projects it's known in advance that a variable should never exceed a certain magnitude (e.g. a variable in repository CASE tool). If it exceeds that magnitude, an error of some sort is present.

Another form of consistency checking that can prove valuable is to compare the measured performance of the result with some predicted performance. This technique is particularly useful in engineering domain where some dynamic system (e.g. software development process) is under

control. The dynamic system can be modeled and the predicted performance obtained from a software implementation of the model. The actual performance of the system can then be measured and compared with model predicted performance. Any significant deviation of measured performance from the predicted performance can indicate a fault.

6.2. Tools Redundancy

The consistency checks technique use extra, or redundant, software to detect faults that can occur in human resource. We have not considered approaches for detecting or possibly tolerating faults that can occur in the software tools used in the SDLC.

Software does not break as human resource does, but instead software faults are the result of incorrect software designs; therefore, any technique that detects faults in software must detect design flaws.

The objective of tools redundancy is to allow certain design flaws in software modules to be detected. The basic concept of tools redundancy is to design and code the software using different tools and to compare the results produced by these tools; besides, the product is designed from the same set of specifications such that each of the modules performs the same function; however, it is hoped that by performing the redundant designs independently, the same mistakes will not be made by the different tools; therefore, when the fault occurs, the fault either does not occur in all modules or it occurs differently in each module, so that the results generated by the modules will differ.

7. Conclusion

As stated in the introduction, our emphasis in this paper is to propose a reliable process model conceptually. This process model is obtained by considering both fault tolerance concepts and software engineering models together. Fault tolerance of development process is accomplished using redundancy which is categorized into three types: human resource redundancy, time redundancy, and software redundancy.

References

[1] J.D. Musa, and W.W. Everett, "Software Reliability Engineering: Technology for the 1990s", IEEE Software, Vol. 7, pp. 36-43, November 1990.

[2] "Handbook of Software Reliability Engineering", McGraw-Hill, editor M.Lyu, 1996.

[3] Sheldaon, F.T., Software Reliability Engineering Case Studies, 8th Intl. Symposium on Software Reliability Engineering, IEEE CS Press, November, 1997.

[4] J.D. Musa, Software Reliability Engineering, McGraw-Hill New York, 1998.

[5] Roger S.Pressman, "Software Engineering", McGrawHill, 2001.

[*6*] Boehm B.W., "Software Risk Management", IEEE Computer Society Press, 1989.

[*7*] Charette R.N., "Software Engineering Risk Analysis and Management", McGrow-Hill 1989.

[*8*] Hall E.M., "Managing Risk: Methods for Software Systems' Development", Addison-Wesley, 1998.

[*9*] Higuera R.P., "Team Risk Management", CrossTalk, U.S. Dept. of Defense, January 1995, p. 2-4.

[*10*] Karolak D.W., "Software Engineering Risk Management", IEEE Computer Society Press, 1996.

[*11*] Keil M., et al, "A Framework for Identifying Software Project Risks", CACM, Vol.41, No.11, November 1998, pp. 76-83.

[*12*] Williams R.C., J.A. Walker, and A.J. Dorofee, "Putting Risk Management into Practice", IEEE Software, May 1997, pp. 75-81.

[*13*] Thomsett R., "The Indiana Jones School of Risk Management", American Programmer, Vol.5, No.7, September 1992, pp. 10-18.

[14] Craig Hollenbach, et al, "Combining Quality and Software Improvement", Communications of ACM, Vol. 40, No.6, June 1997, pp. 41-45.

[15] Meir M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution", Proceeding of the IEEE, Vol. 68, No.9, September 1980.

[16] Taghi M. Khoshgoftaar, Edward B. Allen, Robert Halstead, and Gary P. Trio, "Detection of FaultProne Software Modules During a Spiral Life Cycle", Proceedings of the International Conference on Software Maintenance, Monterey, California, November 4-8, 1996, pp. 69-76.

[17] Lionel C.Briand, Victor R. Basili, and Yong-Mi Kim, "Change Analysis Process to Characterize Software Maintenance Projects", Proceedings of International Conference on Software maintenance, Victoria, British Columbia, Canada, September 19-23, 1994, pp. 38-49.

[18] P. Frankl, R. Hamlet, B. Littlewood, and L. Strigini, "Coosing a Testing Method to Deliver Reliability", International Conference on Software Engineering, pages 68-78, 1997.

[19] Joel Henry, Sallie Henry, Dennis Kafura, and Lance Matheson, "Improving Software Maintenance at Martin Marietta", IEEE Software, Vol. 11, No.4, July 1994, pp. 67-75.

[20] Troy Pearse and Paul Oman, "Maintainability Measurements on Industrial Source Code Maintenance Activities", Proceedings of the International Conference on Software Maintenance, Opio (Nice), France, October 17-20, 1995, pp. 295-303.

[21] Thomas M. Pigoski and Lauren E. Nelson, "Software Maintenance Metrics: A Case Study", Proceedings of the International Conference on Software Maintenance, Victoria, British Columbia, Canada, September 19-23, 1994, pp. 392-401.

[22] Harry Sneed, "Modelling the Maintenance Process at Zurich Life Insurance", Proceedings of the International Conference on Software Maintenance, Monterey, California, November 4-8, 1996, pp. 217226.

[23] George E. Stark, "Measurements for Managing Software Maintenance", Proceedings of the International Conference on Software Maintenance, Monterey, California, November 4-8, 1996, pp. 152161.

[24] Norman F. Schneidewind, "Measuring and Evaluating Maintenance Process Using Reliability, Risk, and Test Metrics", IEEE Transactions on Software Engineering, Vol. 25, No.6, November/Dcember 1999, pp. 768-781.

[25] Johnson B.W., "Fault tolerant microprocessor based systems.", IEEE Micro, Vol.4, No.6, December 1984, pp. 6-21.

[26] Losq J., "A highly efficient redundancy scheme: Self-purging redundancy", IEEE Transactions on Computers, Vol.C-25, No.6, June 1976, pp.569-578.

[27] De Sousa P.T., and F.p. Mathur, "Soft-out modular redundancy", IEEE Transactions on Computers, Vol.C-27, No.7, July 1978, pp. 624-627.

[28] Johnson B.W., J.H. Aylor, and H.H Hana. "Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder", IEEE Journal of Solid-State Circuits, Vol.23, No.1, February 1988, pp. 208-215.

[29] Toy W., N., "Self-checking arithmetic unit", United States Patent Number 4, 314, 350, Bell Telephone Laboratories, Murray Hill, N.J., Feb.2, 1982.