Software Diversity and Fault-Tolerance: An Overview

Daniel Rodriguez Retamosa and Mehrdad Saadatmand
Mälardalen Real-Time Research Centre (MRTC)
Mälardalen University
Västerås, Sweden

dra05002@student.mdh.se , mehrdad.saadatmand@mdh.se

Abstract

The design of reliable and fault-free software is of a major concern for safety-critical real-time and distributed applications. The fault tolerant community addresses these problems through redundancy in hardware components and by diversity, using different software components. Diversity has been used for many years now as a computer defence mechanism to achieve an acceptable degree of fault-tolerance against flaws introduced in design and provides security to software systems. In this paper we give a comprehensive overview of the fault-tolerance techniques based on the design and data diversity approaches. Furthermore, we provide our work with some real applications which implement some of the fault-tolerance methods highlighted within this paper.

Keywords

Design diversity, data diversity, fault-tolerance, dependability

1. Introduction

As a matter of fact, today's real-time and distributed software development faces up to growing system complexity. Nowadays software has to deal with the challenge of enhancing system dependability, meaning its "*ability to deliver service that can justifiably be trusted*" [4] and increasing performance (reliability) in spite of the occurrence of faults within its execution [2,3]. J.M. Voas refers to finding faults at early steps of the software's life-cycle (requirement and design specifications) as an inherent software problem. More than 50% of all failures can be traced back to the specifications. Design faults prevention and removal are not straightforward tasks and thus they can not ensure the absence of faults. In order to shield systems effectively from faults and assure appropriate levels of fault tolerance, not only fault-avoidance techniques are needed but also some other means such us fault treatment and error-processing. The former is based on fault-masking to prevent its activation and the latter aims to substitute erroneous states with error-free states [6]. Persistent faults after the development of safety-critical real-time and distributed systems reveal a lack of dependability and may have catastrophic consequences [1]. The reliability requirements of today's state of the art systems such as flight critical, commercial and military aircrafts hang on the application of strategies to achieve a high degree of fault-tolerance.

Originally, the use of redundant modules was applied on the development of fault-tolerant architectures to effectively deal with physical faults in the hardware of a system. Work in [7] overviews two general architectures: N-Modular Redundancy and M-N Majority Voting, on the basis of redundancy. Those approaches clearly succeed upon fault detection. However, as long as design is a human discipline, systems can not stay away from imperfections and the lack of precise designs may provoke the independent replication of every single fault over system modules. Therefore, simple copies of hardware components, data structures or algorithms are far from being sufficient means to isolate the system from faulty occurrences and prevent a failure of the entire system. More recently, the tolerance of design faults, especially in software, relies on the concept of *diversity* [1,5].

Diversity is an important issue to be considered when building computer systems due to its implications in cost-effectiveness, reliability and safety of fault-tolerance software when it is delivered to operation [13]. The concept can be applied in an extensive way to many applications with safety concerns. Diversity is widely implemented in the area of real-time control in railway, aviation and aerospace industries and in nuclear power plant controls. Moreover, it is also

used to serve in the area of on-line transactions which include bank records and library transactions among others, preventing from failures in data communication [7].

In fault-tolerance software, we find two important approaches within the scope of diversity: *design diversity* and *data diversity* [8]. *Design diversity* is the creation of multiple implementations of a given specification based on the idea that the same fault would not affect all the versions at once, i.e. different implementations have different designs. *Data diversity* is the use of multiple copies of a single implementation with each copy operating on different input data but yielding the same desired results [5].

Popular techniques which are based on the design diversity concept for fault tolerance in software are:

- **Recovery Blocks (RB).** It was first introduced in 1974 by Horning. Early implementations were developed by Randell and Hecht in 1975 and 1981 respectively. Recovery modules (try blocks) run different version of the same algorithm. RB performs fault detection by means of running an acceptance test (AT) on the output of an algorithm. If an AT fails, backward recovery [1] is carried out with aid of a recovery cache and another alternate version is chosen for execution. Many implementations, especially for real-time applications, include a watchdog timer. The RB is categorized as a dynamic redundancy technique [1,11].

- **N-Version Programming (NVP).** It was first suggested by Elmendorf in 1972 and developed by Avizienis and Chen between 1977 and 1978. Compared with the recovery blocks, NVP is a static technique that requires several independent versions of a program for a certain application. These versions execute in parallel and each produces its required output. A voter deems the outputs as acceptable/not acceptable usually via a majority vote. In case of detected errors it performs a forward recovery method to lead the system to a safe state with aid of diverse back-up information [1,10].

Data diversity was introduced by Amman and Knight in [9] after observing that certain failures in a system were caused by combination of specific values in the input. Software programs logically represent input and intermediate data as points at the data space. Unlike design diversity, data diversity uses only one version of the software and applies it to fault-related points at the data space. Therefore fault-tolerance is achieved by using diversity in the data space. In order to complement design diversity in the quest for fault-tolerance software, there exits several data diversity techniques which are similar to the aforementioned for the design diversity approach: retry blocks and N-copy programming [9].

Central sections of this work will go through design and data diversity approaches in detail as means of achieving fault-tolerance in software. Last section provides the reader with an overview of some real applications of diversity.

2. Design Diversity

Generally, faults can exist in different phases of software development (requirements and specifications, design, implementation and coding, testing and maintenance) [1]. But the severity of faults in the first steps of software development can have greater effects than later steps. From another point of view, detection of faults and their effects could be harder also in real-time and embedded systems due to issues like timing, ordering and so forth, than sequential systems. If we consider that complete removal of faults in sequential systems is a hard task, it becomes even 'effectively impossible' for real-time systems [1].

For safety critical systems the continuous and correct operation of the system is very important. Many techniques have been presented to ensure this behavior in these systems. One of the famous ones which is used to increase the fault-tolerance of the system is the use of diversity in the system especially for critical parts. According to [14] the idea behind using diversity is that "two heads are better than one." So if we can implement it in our system and use some kind of *diversity* we can reach the same effect in the system, which is believed to lead to more reliability. For example if one person works on a math problem and reach an answer and another person also comes to the same answer we can have more trust in the correctness of the answer. In short, the main objective for using diversity is to reduce the chance of failure for *independent versions* of a system (or subsystem) due a similar error [1].

Despite this general belief, there are several issues regarding diversity which can highly affect this feature and finally stop it from providing the expected reliability in the system; the most important of these issues is *'dependency'*.

In this section design diversity in software and the related issues will be discussed and two major methods for design diversity are introduced: recovery block and multi version software.

2.1 A Note on Redundant Identical Blocks

Using redundant and identical copies of critical sections of a system has been a common trend in the hardware of a system before its appearance in software development technologies. It was that for critical parts of the system, several copies of a circuit or component were used so that in case the main one stops functioning correctly, another one takes its role and thus the whole system continues to work despite the fault. This technique was quite helpful for detection and tolerance of 'physical faults' [1]. This method can also be used in software fault-tolerance, but there are two major problems with this method in general. The first one is that if there is a problem or defect in the design of such components/circuits, then using identical copies of them is not that helpful as they also contain the same defect. So just using identical backup components is not enough [1]. Another problem about this method is that for real-time systems the time which is required for a backup copy to come to work after a fault is detected with the primary one, could be totally unacceptable; since the same procedure which has finally failed should be repeated in a backup component [7]. (This also applies to non-identical redundant blocks).

2.2 Types of Diversity

From the view point of applying diversity to a system, a system can have *random* or *enforced* diversity. In random diversity, as an example, there may be two (or more) programming teams which are working separately on the same problem and it is just hoped that the created versions do not contain similar problems and thus will not lead to the same errors; i.e. the versions '*fail independently*'. In enforced diversity, the diversity in the system is applied 'systematically' [1], like forcing each group to use a different and specific programming language, algorithm or even data structures to create the software.

2.3 Recovery Blocks

One of the important methods in fault tolerance by design diversity is the use of recovery blocks. In this method several versions of a critical part of a system are made and put in the system to be used sequentially; although it is possible to use them concurrently too [1]. When the system goes through the primary block and realizes that the output is not correct, it moves back to the state before entering that block and tries the same procedure with another block which [usually] provides the same functionality in just a different way. In order to restore a system to move back to the state before entering a block, we need to keep the state of the system in a secure storage called '*recovery cache*' [1]. Also another important part of recovery blocks is the mechanism by which the system detects an incorrect result: *Acceptance Test*. When a block generates an output, acceptance test is performed on it and if an error is found, the system *rolls back* to the previous state and then tries another block with the same input values. This process is repeated until the output of a block passes the acceptance test or there would be no other recovery blocks left to try.

As stated before, this method may not be a good solution for real-time systems in which timing is very important. However, it is still possible to use this method in such systems if a correct estimation can be done about the recovery blocks to find out the amount of time which may be needed to have a correct output value before the deadline, as is shown in Figure 1.

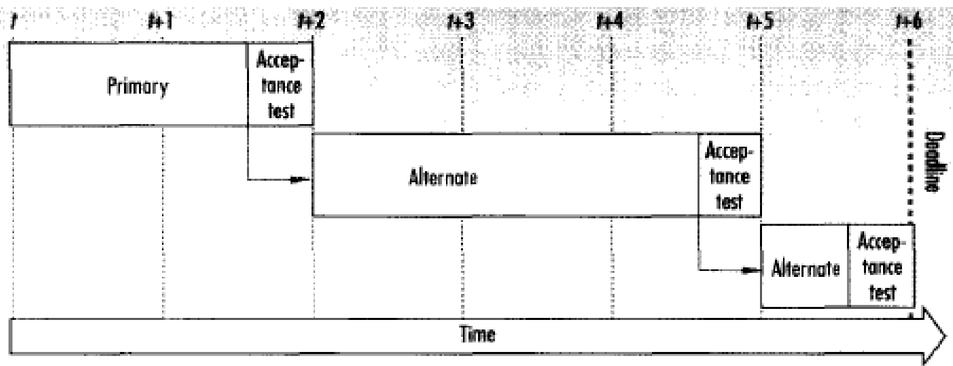

Figure 1: Timing in recovery blocks [1]

A useful diversity for recovery blocks could be the use of different algorithms in different blocks. For example we may use a fast algorithm as our primary block and have an old slower but more reliable one as a backup block for the primary [1]. Since recovery blocks can be time consuming when an error is detected, it is highly recommended that recovery blocks be used for critical parts of the system only.

There are also several important issues and guidelines for acceptance test. First of all it should be remembered that if acceptance test is not designed carefully, it can contain errors too so this part should be kept as simple as possible. The other issue about acceptance test is that performing a reasonable acceptance test could often be time consuming. So some times a faster but less accurate acceptance test could be used instead. For example if the acceptance test should test whether an input array is correctly sorted, one solution could be to check the order of the items in the array (ascending or descending) and also checking the existence of all the input items in the output which can be very time consuming. An alternative to provide the same testing in shorter time could be checking the order of items and then instead of performing a test to see if all items are present, it is possible to check the sum of the items in the output array against the sum of the items in the input which is a little less accurate but faster [1].

With careful time estimation, parallel execution of recovery blocks is also possible. In this scenario, blocks could have different execution times and the output is not the one which is produced by the fastest block, but it will be the output of the block which has the highest priority compared to others that has also passed its acceptance test. In other words, recovery blocks are executed in parallel and by the deadline time (or when all executions are finished), among all generated correct outputs, the one which belongs to the block with highest priority will be selected [1].

In systems where there are concurrent processes that communicate with each other, usage of recovery blocks would require more considerations since the effects of communication between processes should also be taken into account. So if at one point an error is detected roll back should be done until the point where the effect of communication between processes is also neutralized. In order to reduce these roll backs for communications, it is possible to create a '*restore point*' just as soon as some processes start a *conversation* and there should be no communication between these processes with other processes which are not in the conversation. An important point for this case is that processes in a conversation should finish it together to reduce the 'domino effect' [1] of rollbacks.

2.4 Multiversion (N-Version) Programming

In this method all versions are run in parallel and usually on different machines. It is one of the important methods for systems which require fault-tolerance in both hardware and software parts. It is also possible that on systems which do not have necessary requirements to run them in parallel, versions are run sequentially. The key point in this method is that after the execution of all versions the results are gathered and then the system decides upon the correct result by consensus. The part of the system which is responsible for this task is the decision mechanism (or voting mechanism). An illustration of how a multiversion system works is shown in the following picture:

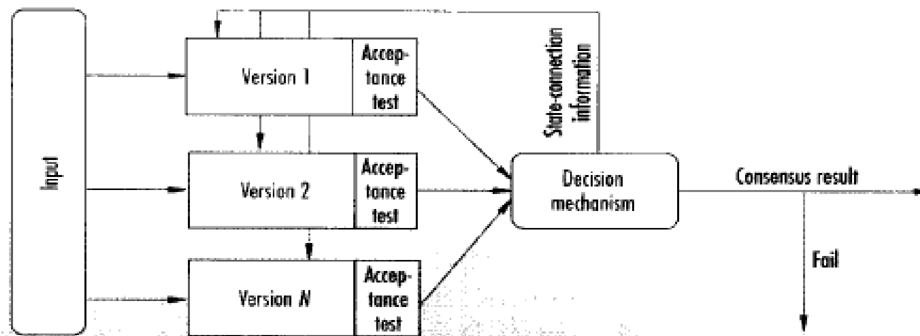

Figure 2: Multiversion architecture [1]

In some implementations of mutliversion systems, an acceptance test is also added to each version to filter out wrong results from the decision mechanism. The difference between the type of acceptance test used here with that of recovery blocks is that here it is not needed to be a very complicated acceptance test since its goal is only to filter out obvious wrong results or put away versions which have not produced results and help the system not wait unreasonably for such versions to send their results [1].

Another good feature which can be implemented in this method is having a feedback. So after the decision mechanism decides upon the final result, the result will be sent to those versions which provided a wrong output. This way these parts can adapt and reconfigure themselves to generate correct answers for next operations (and rollback to the correct state). When two versions are used, the system is able to detect when an error has occurred in the system but there is no way to judge which could be the correct answer. In order to 'mask the error' in the system at least three or more versions are needed. The decision mechanism itself is likely to contain errors, so designing this part is very important and it must be quite reliable. It is also possible to use diversity for decision mechanism [1]. The main difference between decision mechanism and acceptance test which is used in recovery blocks is that, acceptance test works on a single output at a time (individually) while decision mechanism receives all the results and then works on them. Therefore timing in systems which use decision mechanisms is very important and all versions should finish their operation and submit their results in a reasonable time. Due to this fact in real-time systems usually a specific amount of time is allowed for versions to hand in their results, otherwise the decision mechanism ignores them and starts its operation on the already received answers from other versions. This specific time should be decided according to the deadline of the task.

According to [1] it is possible to categorize the results of acceptance test in each version in four groups: G (good) which mean the result of a version has passed its acceptance test and it is also a correct value , D (detected error) when an acceptance test realizes an incorrect value and so it does not enter the decision mechanism, U (undetected error) means the acceptance test considered it as correct so it has passed the acceptance test and entered decision mechanism but it is not a correct value according to the problem, and finally S (similar errors) refers to those incorrect values which are similar between different versions (which may lead to a consensus!). Different combination of these outputs leads to different final results of the system. For example in case of DDU in a three version system, since the system has detected two incorrect values(DD) so the only one on which it can work is U. In such systems if it is allowed to produce the final answer by just having one value then the output of the system will be that incorrect value (U). It is possible to stop this effect by not letting the system "degrade to a simplex mode" [1] instead of triplex (as an example).

An important fact about decision mechanism which should be remembered during design is that in some situations it could be just more than a simple voting and consensus system and according

to the type or results it is dealing with, the voting and consensus mechanism in it could be a lot different and more complex. For example in programs that produce a real or a string value the voting mechanism could be somewhat different, as in string values, there could be several representations of a specific word or sentence which differ in usage of upper/lower case characters, spacing and punctuation. Also in real values for example there could be some minor differences between produced results from different versions in the sixth or seventh digit after the decimal point. So according to the context and purpose of the program, decision mechanism maybe implemented differently for such situations whether to consider these cases as similar or not similar. Another example is shown in Figure 3. As is shown in that picture, different versions produce pair values and the decision mechanism should reach a consensus on these results. If only pairs are important to us as a single unit then no consensus can be made on them as they are not the same, but it can be seen that there are common values in the pairs which could mean a consensus if they are considered from a different aspect. So in the right diagram in the figure, if common generated values are considered, it is possible to select (A,B) as the final result in the decision mechanism as two versions have produced A and two have produced B.

A different architecture for diversity by NVP (N-version programming) is "*N Self Checking Programming*" or NSCP in short. In this architecture versions are grouped into pairs which help them verify their own results. In terms of hardware, each pair could run on the same machine in this model. The general design is shown in the Figure 4. You can read more about that model in [15].

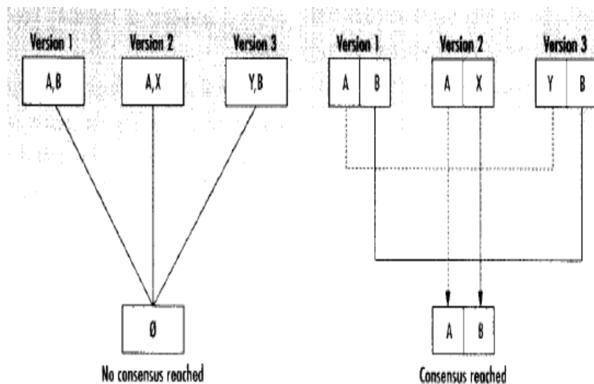

Figure 3: Different decision mechanisms according to the context [1].

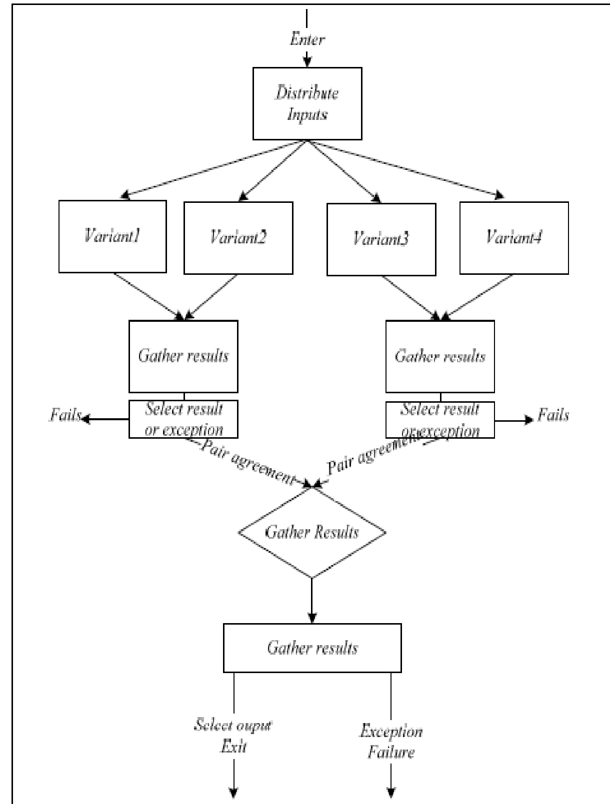

Figure 4: N Self-Checking Programming [15].

2.5 Diversity Issues

In previous sections, diversity methods were discussed and some important considerations in using each of them mentioned. In this part, some major problems in software diversity in general, are discussed.

One of the major issues with diversity is the cost. Although diversity can provide a good approach to make the system fault tolerant and mask errors, but it can dramatically increase the cost of a project. According to [14] this increase is even not linear with the versions. When different versions are produced, each version adds its own additional costs to the system, plus potential integration and design costs which are automatically added when n-version programming is used. Considering design, it should be remembered that designing a redundant system requires more work than just a normal non-redundant one as some issues should be taken into account in the design phase. More work in requirement part is also needed as they should be

defined in a way to be suitable for all the versions. Although this feature may lead to having better and mode detailed requirements, but of course it means more work and cost. Also after creating each version, testing of each version is also important. Here it is suggested that instead of testing each version separately, a back-to-back[1] test be done to reduce the cost of testing each version individually. To run different versions of the program, different machines and processors are usually used and this also adds its own expenses. Therefore, the final cost of the project should also be considered in comparison to the reliability gained by using diversity.

The other issue in software diversity which is very important is the problem of independence. As mentioned in previous sections the main goal of diversity is to have different parts in the system which act differently on the same input to produce the expected output. So if there is a bug somewhere in one version, because of different implementation and design of other ones, those other versions do not fail on that input so the system will continue to work and the error is masked in the system. This behavior requires an important factor which is independency. Versions should be made independent of each other so that no similar bug exists in them. As stated, diversity is applied in the system either randomly or systematically (enforced). Both of these methods follow the goal of independency. Independency can be achieved by having different programming teams using different programming languages, algorithms and data types, etc.

There are several reasons for having similar bugs in the system. Mostly a problem in the design or requirements can lead to all independent systems following and implementing it and thus all producing the same error. Independency in implementation is also very important. An example for a 'similar' mistake which can occur in implementation of different versions can be problems related to out of boundary issues which are possible to be neglected easily [1].

[1] Back-to-Back Test: In back to back testing, all versions are presented with the same input and if there is any discrepancy found in the results of the versions, then the versions are examined to see which ones have produced the correct and incorrect results and then the erroneous versions are fixed accordingly.

One key fact about making similar mistakes is that some problems are inherently harder in nature than others and thus no matter if different independent implementations for it are being developed, there is a high risk that independent teams make the same mistake on that particular hard part. So "variation of difficulty over the demand space" [14] is a very important cause for having similar mistakes among independent versions and should be taken into account to provide better independency and diversity.

In short, the final purpose of analyzing all these pitfalls is to achieve a design in which all versions fail independently given different inputs, if they are going to fail.

3. Data diversity

In previous sections we have described the general concept of design diversity as a technique for software fault tolerance and have covered the recovery block and N-version programming as two possible approaches to its implementation.

The data diversity approach relies on the observation that software programs sometimes fail for certain values in the input space and these failures could be averted if there is a minor perturbation of input data which is acceptable to the software, i.e. diversity in the data space may avoid sequences of events that lead to failures. For instance, sensors always provide precise data and small modifications to those data would affect the application [9]. The point with data diversity is to achieve fault-tolerance by re-interpreting the input data and producing data points out of well defined *failure domain* boundaries for the program [9]. A failure domain is defined in [12] as a set of input values that cause program failures.

The central part of any data diversity scheme is thus the *re-expression algorithm* which is in charge of transforming the input in a re-expressed input, i.e. it represents the original input in a different form. As basic examples of data re-expressions we could mention changes to floating point values (lose precision), reordering of data sequences, changes in data timing, reordering of transactions or rewrite SQL code among many others.

Depending on the density of the failure domain, the re-expressed input may lie out of the failure

boundaries. Data diversity uses identical copies of one version of software for a certain specification. The program executes correctly if and only if the re-expression of the input is not within the failure domain [9]. The re-expression of inputs can be exact or just simple approximations to the information in input. *Exact re-expression* works well for error detection but they are less flexible to avoid the initial causes of failure [9]. On the other hand, approximated re-expressions are easier to generate and more likely succeed in avoiding the failure region. Given the importance the re-expression algorithm has for the data diversity approach, it becomes essential to keep its design free of faults. Obviously, simple re-expression algorithms are easier to implement than more complex ones and therefore they may contain fewer design faults [9].

3.1 Retry blocks

The *retry blocks* scheme crops up as an adaptation of recovery blocks scheme to use data diversity. Re-expressed data is forwarded to a retry block which runs the same version of the software replicated at each block. Their outputs are evaluated at a retry block's acceptance test which deems the validity of a certain result of the algorithm. A retry block's acceptance test is equivalent to a recovery block's acceptance test [9]. The retry blocks technique first attempts to pass the acceptance test by using the primary algorithm. If for any reason the result of primary algorithm result fails the acceptance test, then a re-expressed form of input data will be given to the same algorithm and the algorithm is executed again with this new data until a correct output is produced (retry block complete) or the process's deadline is violated [9,15]. If the deadline expires, a backward recovery-based backup algorithm is invoked with the original input data. The backup algorithm is also evaluated by the acceptance test and an error exception is generated if this backup algorithm is not successful.

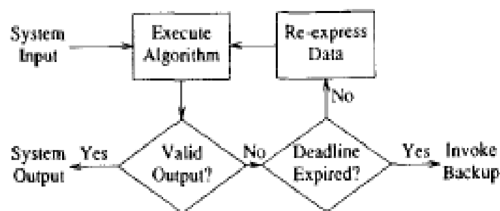

Figure 5: Retry Blocks Structure [9]

Finally it is important to know that the performance of the acceptance test determines the performance of the retry blocks technique [9].

3.2 N-copy programming

The *N-copy programming* scheme is the data diverse complement of N-version programming for the design diversity approach. This technique uses a voting mechanism to select the correct output and a forward recovery algorithm to obtain fault-tolerance. In practice, N copies of the same program execute concurrently, each on a set of data produced by re-expression but sequential execution is also achievable with data diversity. Re-expressed data sets are distributed between copies of the program and outputs are provided to the decision mechanism. Outputs generated by the different copies can converge or diverge depending on which re-expression algorithm is selected.

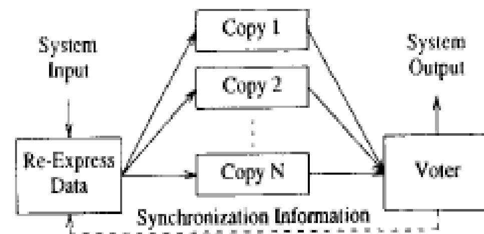

Figure 6: N-copy Programming Structure [9]

Exact algorithms should generate identical outputs and the final result can be obtained by simple majority voting. On the other hand, approximate algorithms may produce different but still acceptable outputs, however unfortunately a decision based on majority is not applicable in this situation, though. The final system output in this case is selected according to the frequency of occurrence of intermediate outputs (calculated on re-expressed inputs). For example, if a certain approximation of the exact correct output is obtained several times, the likelihood of that being selected is bigger. If the number of occurrences of all outputs is equal, then an arbitrary choice is to be carried out [9].

4. Diversity in practice

In this section we explore a real-world example of using diversity within the area of real-time control applications in aviation and space applications.

Space shuttle

This is an excellent example of the concept of design diversity applied to real-time control on the NASA's Space Shuttle, where design faults in the system were tried to be avoided by using diversity [7].

Five computers carry out all necessary guidance and flight control operations. The Shuttle's primary system consists of four computer running identical control software versions. In addition to this, each computer is fed with the output of the other three computers and performs a validation check of those outputs by comparing its own output with those data via software. Furthermore, every single computer is able to report errors sensed on any of the other three computers by forwarding an error message through an error flag line. A faulty computer interrupts its operations if a voting on the three error lines results to be positive, and informs the rest about the exceptional situation. The overall system bears the occurrence of maximum two failures. A sequence of two failures forces the system to run in a duplex mode of operation. The two remaining primary computers attempt to avoid the occurrence of a third failure performing self-testing and comparison of their outputs. If it can be avoided, the system runs in an uncertain state [7].

The fifth computer implements a different version of the design and assists the primary system by performing validation actions over the outputs of the primary computers in order to guarantee that failures can not be caused by common operation bugs such as identical, but incorrect outputs produced by those four primary computers [7].

5. Conclusions

In this paper different aspects of diversity in software development were covered. Two major types of diversity were introduced: design diversity and data diversity. Design diversity can be achieved by using recovery blocks or different versions attempting to solve a specific problem. Also we saw that although diversity can help us build more reliable applications but on the other hand it can increase significantly the final cost of the project. The other main issue which we discussed and should be taken into account is the issue of independency which is the main idea behind design diversity.

6. References

[1] John P.J Kelly, Thomas I. McVittie, Waine I. Yamamoto, "Implementing design diversity to achieve fault-tolerance", IEEE 0740-7459/91/0700/0061

[2]Rogério de Lemos, José Luiz Fiadeiro, "An Architectural Support for Self-Adaptive Software for Treating Faults", *WOSS '02*, Nov 18-19, 2002, Charleston, SC, USA. 2002 ACM 1-58113-609-9/02/0011

[3] Pankaj Jalote, Satish K. Tripathi, "Final Report on Workshop on Integrated Approach for Fault Tolerance - Current State and Future Requirements"

[4] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, "Fundamental Concepts of Dependability", Research Report No 1145, LAAS-CNRS, April 2001.

[5] Avizienis, A. Kelly, J.P.J., "Fault Tolerance by Design Diversity: Concepts and Experiments" IEEE, Aug. 1984, Vol 17, Issue: 8, page(s) 67- 80, ISSN: 0018-9162

[6] J.C Laprie et al. "Dependability: Basic Concepts and Terminology", IFIP WG 10.4 – Dependable Computing and Fault Tolerance, Aug 94.

[7] Mohamad R. Neilforoshan, "Fault tolerant computing in computer design"

[8]"A Survey of Software Fault Tolerance Techniques"; Zaipeng Xie, Hongyu Sun, Kewal Saluja.

[9] "Data Diversity: An Approach to Software Fault Tolerance", R. E. Ammann and J. C. Knight, IEEE Transactions on Computers, April 1988 (Vol. 37, No. 4)

[10] A. Avizienis, "The N-version approach to fault-tolerant software," IEEE Trans. Software Eng., vol. SE-11, Dec. 1985

[11] B. Randell, "System structure for software fault tolerance," *IEEE* Trans. Software Eng., vol SE-I, June 1975

[12] F Cnstian, "Exception handling," in *Resilient Computing Systems, Vol. 2,* T Anderson, Ed New York

[13] B. Littlewood, L. Strigini, "Fault tolerance via diversity against design faults: design principles and reliability assessment", ICSE 2000, Limerick, Ireland, ACM 23000 1-58113-206-9/00/

[14] BEV LITTLEWOOD, PETER POPOV, and LORENZO STRIGINI, "Modeling Software Design Diversity – A Review", Centre for Software Reliability, City University

[15] Zaipeng Xie, Hongyu Sun and Kewal Saluja, "A SURVEY OF SOFTWARE FAULT TOLERANCE TECHNIQUES"