Software Fault Tolerance

Kazimov Tofig Hasanaga, Jalilian Shahrukh Mostafa

Azerbaijan National Academy of Sciences Institute of Information Technology, A31141, Baku,
Ph. Agayev st. 9, E-mail: depart9@iit.ab.az

Abstract − Because of our present inability to produce error-free software, software fault tolerance is and will continue to be an important consideration in software systems. The root cause of software design errors is the complexity of the systems. This paper surveys various software fault tolerance techniques and methodologies. They are two groups: Single version and Multi version software fault tolerance techniques. It is expected that software fault tolerance research will benefit from this research by enabling greater predictability of the dependability of software.

I. Introduction

In this paper we present fault tolerance techniques applicable to software. These techniques are divided into two groups: Single version and Multi-version software techniques. Single version techniques focus on improving the fault tolerance of a single piece of software by adding mechanisms into the design targeting the detection, containment, and handling of errors caused by the activation of design faults. Multi-version fault tolerance techniques use multiple versions of a piece of software in a structured way to ensure that design faults in one version do not cause system failures. A characteristic of the software fault tolerance techniques is that they can, in principle, be applied at any level in a software system: procedure, process, full application program, or the whole system including the operating system [1].

II. Single –Version Software Fault Tolerance Techniques

Single version techniques add to a single software module a number of functional capabilities that are unnecessary in a fault-free environment. Software structure and actions are modified to be able to detect a fault, isolate it and prevent the propagation of its effect throughout the system. In this section, we consider how fault detection, fault containment and fault recovery are achieved in software domain [3]. The schema of the complete taxonomy of the single version software fault tolerance techniques is shown in Figure 1.

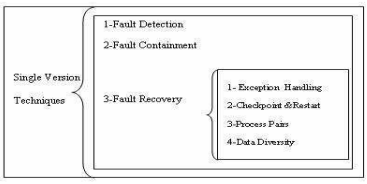

Figure 1.single – version software fault tolerance techniques

II. 1. Fault Detection Techniques

The goal of fault detection in software is to determine that a fault has occurred within a system. Single-version fault tolerance techniques usually use various types of acceptance tests to detect faults. The result of a program is subjected to a test. If the result passes the test, the program continues its execution. A failed test indicates a fault. A test is most effective if it can be calculated in a simple way and if it is based on criteria that can be derived independently of the program application. The existing techniques include timing checks, coding checks, reversal checks, reasonableness checks and structural checks[3].

Timing checks; are applicable to systems whose specification include timing constrains. Based on these constrains, checks can be developed to indicate a deviation from the required behavior. Watchdog timer is an example of a timing check. Coding checks; are applicable to systems whose data can be encoded using information redundancy techniques. Cyclic redundancy checks can be used in cases when the information is merely transported from one module to another without changing its content. Arithmetic codes can be used to detect errors in arithmetic operations. In some systems, it is possible to reverse the output values and to compute the corresponding input values. For such system, reversal checks; can be applied. A reversal check compares the actual inputs of the system with the computed ones. A disagreement indicates a fault. Reasonableness checks; use semantic properties of data to detect fault. For example, a range of data can be examined for overflow or underflow to indicate a deviation from system's requirements. Structural checks; are based on known properties of data structures. For example, a number of elements in a list can be counted, or links and pointers can be verified. Structural checks can be made more efficient by adding redundant data to a data structure, e.g. attaching counts on the number of items in a list, or adding extra pointers.

II. 2. Fault Containment Techniques

Fault containment in software can be achieved by modifying the structure of the system and by putting a set of restrictions defining which actions are permissible within the system. In this section, we describe four techniques for fault containment: modularization, partitioning, system closure and atomic actions [3].

It is common to decompose a software system into modules with few or no common dependencies between them. Before performing modularization, visibility and connectivity parameters are examined to determine which module possesses highest potential to cause system failure. The isolation between functionally independent modules can be done by partitioning the modular hierarchy of a software architecture in horizontal or vertical dimensions. Another technique used for fault containment in software is system closure. This technique is based on a principle that no action is permissible unless explicitly authorized. In an environment with many restrictions and strict control all the interactions between

the elements of the system are visible. Therefore, it is easier to locate and remove any fault. An alternative technique for fault containment uses atomic actions to define interactions between system components. An atomic action among a group of components is an activity in which the components interact exclusively with each other. There is no interaction with the rest of the system for the duration of the activity. Within an atomic action, the participating components neither import, nor export any type of information from non-participating components of the system.

1. Fault Recovery Techniques

Once a fault is detected and contained, a system attempts to recover from the faulty state and regain operational status. If fault detection and containment mechanisms are implemented properly, the effects of the faults are contained within a particular set of modules at the moment of fault detection. The knowledge of fault containment region is essential for the design of effective fault recovery mechanism [3].

II.2.1. Exception Handling

In many software systems, the request for initiation of fault recovery is issued by exception handling. Exception handling is the interruption of normal operation to handle abnormal responses. Possible events triggering the exceptions in a software module can be classified into three groups [3]: Interface exceptions; are signaled by a module when it detects an invalid service request. This type of exception is supposed to be handled by the module that requested the service. Local exceptions; are signaled by a module when its fault detection mechanism detects a fault within its internal operations. This type of exception is supposed to be handled by the faulty module. Failure exceptions; are signaled by a module when it has detected that its fault recovery mechanism is enable to recover successfully. This type of exception is supposed to be handled by the system.

II.2.2. Checkpoint and Restart

A popular recovery mechanism for single-version software fault tolerance is checkpoint and restart, also referred to as backward error recovery. As mentioned previously, most of the software faults are design faults, activated by some unexpected input sequence. These type of faults resemble hardware intermittent faults: they appear for a short period of time, then disappear, and then may appear again. As in hardware case, simply restarting the module is usually enough to successfully complete its execution. The general scheme of checkpoint and restart recovery mechanism is shown in Figure 2.

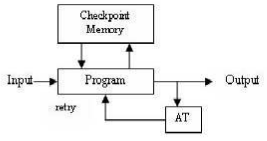

Figure 2. Checkpoint and Restart Recovery

II.2.3. Process Pairs

A process pair uses two identical versions of the software that run on separate processors (Figure 3). The recovery mechanism is checkpoint and restart. Here the processors are labeled as primary and secondary. At first the primary processor is actively processing the input and creating the output while generating checkpoint information that is sent to the backup or secondary processor. Upon error detection, the secondary processor loads the last checkpoint as its starting state and takes over the role of primary processor. As this happens, the faulty processor goes offline and executes diagnostic checks. If required, maintenance and replacement is performed on the faulty processor. After returning to service the repaired processor becomes the secondary processor and begins taking checkpoints from the primary[1].

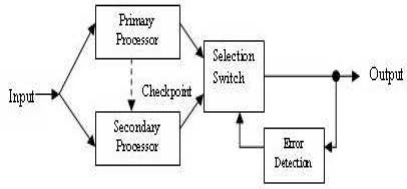

Figure 3. Logical Representation of Process Pairs

II.2.4. Data Diversity

Data diversity is a technique aiming to improve the efficiency of checkpoint and restart by using different inputs re-expressions for each retry. Therefore, if inputs are re-expressed in a diverse way, it is unlikely that different re-expressions activate the same fault[3]. Data re-expression is used to obtain alternate input data by generating logically equivalent input data sets. Given initial data within the program failure region, the re-expressed input data should exist outside that failure region. A re-expression algorithm, R, transforms the original input x to produce the new input, y = R(x). The input y may either approximate x or contain x's information in a different form. The program, P, and R determine the relationship between P(x) and P(y). Figure 4 illustrates basic data re-expression[2].

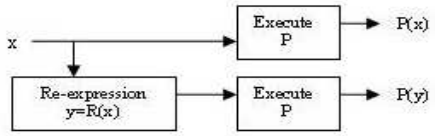

Figure 4. Basic Data Re-expression method.

There are three basic techniques for data diversity [1]:

- Input Data Re-Expression, where only the input is changed.
- Input Re-Expression with Post-Execution Adjustment, where the output is also processed as necessary to achieve the required output value or format.
- Re-Expression via Decomposition and Recombination, where the input is broken down into smaller elements and then recombined after processing to form the desired output.

III. Multi –Version Software Fault Tolerance Techniques

Multi-version techniques use two or more versions of the same software module, which satisfy the design diversity requirements. For example, different teams, different coding languages or different algorithms can be used to maximize the probability that all the versions do not have common faults. This section covers some of these "design diversity" approaches to software reliability and safety(see figure 5).

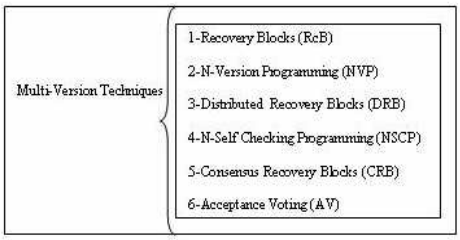

Figure 5. Multi –version software fault tolerance techniques

III.1. Recovery Blocks (RcB) Techniques

The Recovery Blocks technique combines the basics of the checkpoint and restart approach with multiple versions of a software component such that a different version is tried after an error is detected (see Figure 6). Checkpoints are created before a version executes. Checkpoints are needed to recover the state after a version fails to provide a valid operational starting point for the next version if an error is detected. The acceptance test need not be an output only test and can be implemented by various embedded checks to increase the effectiveness of the error detection. Also, because the primary version will be executed successfully most of the time, the alternates could be designed to provide degraded performance in some sense.

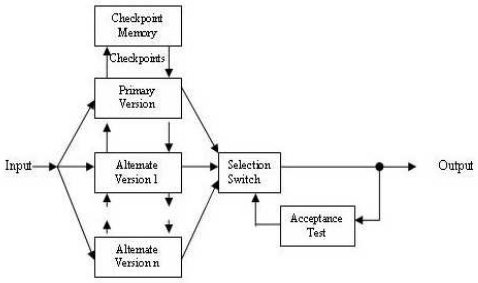

Figure 6. Recovery Block Model

III.2. N-Version Programming (NVP) Techniques

The N-version programming techniques resemble the N-modular hardware redundancy. The block diagram is shown in Figure 7. It consists of n different software implementations of a module, executed concurrently. Each version accomplishes the same task, but in a different way. The selection algorithm decides which of the answers is correct and returns this answer as a result of the modules execution. The selection algorithm is usually implemented as a generic voter. This is an advantage over recovery block fault detection mechanism, requiring application dependent acceptance tests.

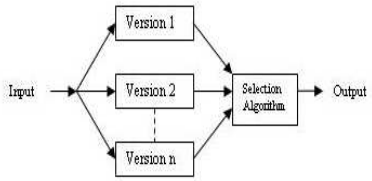

Figure 7. N-Version Programming Model

III.3. N-Self Checking Programming (NSCP) Techniques

N-Self Checking programming combines recovery blocks concept with N version programming. The checking is performed either by using acceptance tests, or by using comparison. N self-checking programming using acceptance tests is shown in Figure8. Different versions of the program module and the acceptance tests AT are developed independently from common requirements. The individual checks for each of the version are either embedded in the code, or placed at the output. The use of separate acceptance tests for each version is the main difference of this technique from recovery blocks approach. The execution of each version can be done either serially, or concurrently.

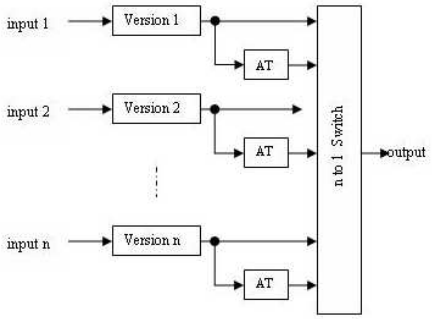

Figure8. N Self-Checking Programming using Acceptance Tests

III.4. Distributed Recovery Blocks (DRB) Techniques

The DRB technique is a combination of distributed and/or parallel processing and recovery blocks that provides both hardware and software fault tolerance. Emphasis in the development of the technique has been placed on real-time target applications, distributed and parallel computing systems, and handling both hardware and software faults. Although DRB uses recovery blocks, it implements a forward recovery scheme, consistent with its emphasis on real-time applications.

The techniques architecture consists of a pair of self-checking processing nodes (PSP). The PSP scheme uses two copies of a self-checking computing component that are structured as a primary-shadow pair, resident on two or more networked nodes(see figure 9).

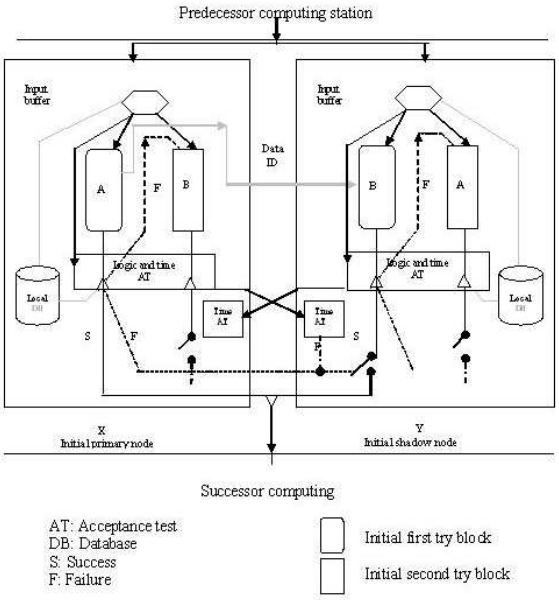

Figure 9. Distributed Recovery Block structure.

III.5. Consensus Recovery Blocks (CRB) Techniques

The CRB technique, combines RcB and NVP implementation techniques. CRB uses n variants, as in RcB, in order of their service and reliability. The n variants are first run concurrently in NVP fashion, and their results are checked by a voter.

If the voter does not determine a correct result, then the results of the highest ranked variant are submitted to the AT. If that variant's results fail the AT, then the next highest ranked variant's results are sent to the AT, and so on, until an acceptable result passes the AT or no variants are left. In the RcB part of the CRB technique, the existing results of variant execution, that is, the ones that just failed to result in a majority decision, can be run through the AT, or, if a transient failure is likely, the variants can be run again prior to submitting their results to the AT.

III.6. Acceptance Voting (AV) Techniques

The AV technique uses both an AT and a voting-type DM, along with forward recovery to accomplish fault tolerance. In AV, all variants can execute in parallel. The variant results are evaluated by an AT, and only accepted results are sent to the voter. Since the DM may see anywhere from 1 to n (where n is the number of variants) results, the technique requires a dynamic voting algorithm. The dynamic voter is able to process a varying number of results upon each invocation. That is, if two results pass the AT, they are compared. If five results pass, they are voted upon, and so on. If no results pass the AT, then the system fails. It also fails if the dynamic voter cannot select a correct result.

IV.Conclusions

In this paper we have presented a review of software fault tolerance. We gave a brief overview of the software development processes and noted how hard-to-detect design faults are likely to be introduced during development. For some applications software safety is more important than reliability, and fault tolerance techniques used in those applications are aimed at preventing catastrophes. Because of our present inability to produce error-free software, software fault tolerance is and will continue to be an important consideration in software systems. Current research in software engineering focuses on establishing patterns in the software structure. It is expected that software fault tolerance research will benefit from this research by enabling greater predictability of the dependability of software.

References

[1] Wilfredo Torres-Pomales, Software Fault Tolerance: A Tutorial, NASA, Langley Research Center, 2000.

[2] Laura L. Pullum, Software Fault Tolerance Techniques and Implementation, Artech House, INC., Norwood, 2001.

[3] Elena Dubrova, Fault Tolerant Design; An Introduction Draft., Kluwer Publishers, London, 2007.

[4] Hecht H., Fault Tolerant Software for Real-Time Applications, ACM Computing Surveys, Vol.8, No.4.

[5] Zaipeng Xie, A Survey of Fault Tolerance Techniques, University of Wisconsin-Madison.

[6] Avizienis,A.,Fault Tolerance by Design Diversity, IEEE Computer.

[7] K.H. KIM, The Distributed Recovery Block Scheme, university of California, Software Fault Tolerance, 1995.

[8] Brian Randell, Software Fault Tolerancy, University of Newcastel Tyne, John Wiley & Sons Ltd, England,1995.