The Multi-Layered Design Diversity Architecture:
Application of the Design Diversity Approach
to Multiple System Layers

Aki Watanabe Hiroaki Takada Ken Sakamura

Department of Information Science,
Faculty of Science, University of Tokyo.
7-3-1, Hongo, Bunkyo-ku, Tokyo, Japan

Abstract

This paper describes the multi-layered design diversity (MLDD) architecture which achieves fault tolerance to design faults of application programs, operating systems, and hardware components through applying the design diversity approach to these three system layers.

The introduction of design diversity into multiple system layers improves system reliability. However, its enormous amount of cost has it impractical. We solve this problem through the fact that the concept of the TRON Project standardization approach to achieve compatibility among systems is same with that of the design diversity approach.

In order for the MLDD architecture to be effective in improving system reliability, a probability of a coincident error, that is, two or more independently developed implementations failing on the same input, must be low. We conjecture that a low coincident error rate can be achieved using sufficiently high quality development procedures for real-life applications and different testing methods for developing multiple implementations. This paper also describes an ongoing examination of this conjecture.

1 Introduction

In the Highly Functionally Distributed Systems (HFDS), which the TRON Project [1] aims to realize, many thousands of computers play an important role in every aspect of our living environments. As a result, if computer systems controlling traffic fail, many lives are lost. In HFDS, the consequence of system failures has significant impact [2].

Previously, physical faults of hardware components were the dominant causes of system failures. However, as hardware components become less susceptible to physical faults and the size and the complexity of systems steadily increase, software design faults, so called bugs, are becoming more common causes of system failures.

For tolerating physical faults, the use of redundant copies of hardware components are quite effective. However, since design faults are reproduced when redundant copies are made, simple replication of hardware or software elements is insufficient for design fault tolerance. Instead, design fault tolerance is achieved using the design diversity approach [3], in which implementations for multiple computations are not copies but independently developed to meet a common specification.

Nearly all the research on the design diversity approach has been carried out is for software, especially for application programs. In many safety-related systems which uses the design diversity approach, programs independently developed from a common specification are executed on hardware components from different suppliers. For instance, the flight control system of the Airbus Industry A320 aircraft [4] uses two diverse programs and two types of duplex computer composed of two computation channels. Each duplex computer executes the two programs with one channel executing one program.

The previous approach to design fault tolerance has some issues that require resolution in order to achieve ultra-high reliability. For example, it is impossible to recover from errors due to design faults in operating systems and hardware elements [5]. Moreover, since one hardware element is not compatible with the other ones, when a hardware failure occurs, not only the failed hardware component but also the program executed on it is removed from service.

Today, the growing complexity of VLSI circuits
makes a complete verification of the design very difficult. This results in the increase of the number of faults in hardware component design. The more frequently VLSI circuits are introduced into safety-critical systems, the more powerful design fault tolerance ability is needed. Requirements for a new approach to design fault tolerance are increasing.

This paper proposes the multi-layered design diversity (MLDD) architecture as a new approach to design fault tolerance. The MLDD architecture, divided into three system layers of an application program layer, an operating system layer, and hardware layer, applies the design diversity approach to these system layers with specifications used for developing implementations of each system layer. The MLDD architecture can implement recovery from errors due to design faults of operating systems and hardware components, and can realize repair of a failed implementation with a minor impact on system reliability.

The application of the design diversity approach to multiple system layers improves system reliability. However, its inordinately high cost has made it impractical. In the TRON Project, for achieving compatibility among systems, any number of manufacturers are allowed to implement CPU chips or operating systems on the common specification. Today, several operating systems developed independently from the same specifications such as μITRON, ITRON1 and ITRON2 [6, 7] and multiple CPU chips based on the TRON CPU specification [8] exist. This enables us to investigate on the MLDD architecture without regard to the cost.

A probability of coincident errors determines the effectiveness of the MLDD architecture. In order to estimate coincident error rate, we are carrying out the examination of residual faults of several operating systems based on the ITRON2 specification and of CPU chips based on the TRON CPU specification. We will also describe this ongoing examination.

2 Related Work

A number of papers have been devoted to software fault tolerance. The most widely known methods are recovery-block [9] and N-version programming [10], which use multiple programs independently developed from the same specifications.

The recovery-block scheme is analogous to the hardware fault tolerance technique called stand-by sparing. This scheme consists of multiple application programs called alternates and an error detection routine called an acceptance test. After the primary alternate completes its execution, its results are subjected to the acceptance test. If they do not satisfy the acceptance test, the application program state is restored to that before the execution of the primary alternate and the secondary alternate is executed. This process is repeated until either the acceptance test is passed or no more alternates are available.

The N-version programming strategy is analogous to the hardware fault tolerance technique called N-modular redundancy. In this approach, multiple application programs are executed in parallel, and majority voting on their results selects the results to be used. Several safety-related systems such as those in the NASA Space Shuttle [11] and the Airbus A-320 [4] achieve software-fault tolerance by using the N-version programming strategy with two diverse application programs.

The Space Shuttle computing system contains five identical general purpose digital computers and two diverse application programs. During critical mission phases, four computers execute identical application programs, and the fifth executes the other application program. Hardware designs are assumed to be correct. The A320 has two types of computer and two diversified programs with each computer executing both programs. Four different programs are required because one computer is not compatible with the other one. In both systems, a failed computer is removed from service and is not recovered to a correct state.

3 Concepts of the MLDD Architecture

The multi-layered design diversity (MLDD) architecture is the three-layered system structure in which the design diversity approach is applied to three layers of application programs, operating systems, and hardware components with specifications used for developing implementations of each layer (Figure 1).

In the rest of this paper, a component means an implementation which is developed from a specification for a system layer. Each of three system layers has multiple components. In Figure 1, the hardware layer possesses three components. We refer to an execution unit consisting of an application program component, an operating system component, and a hardware component as a unit.

The major feature of the MLDD architecture is compatibility among components of each layer. This is because the components of each layer are developed from a common specification. In the MLDD architec-
4 Fault Tolerance Functions of the MLDD Architecture

The adoption of the design diversity approach achieves the required system reliability using less reliable units. This is because the failure rate of multiple diverse units is lower than the failure rate of any individual unit. Since the probability of failure of a small number of diverse units is high, reduction of the number of diverse units degrades system reliability. Therefore, for attaining high system reliability, not reducing the redundancy of units is important. For this purpose, the MLDD architecture implements recovery from errors due to design faults of operating systems and of hardware components and identification of a faulty component.

4.1 Fault Recovery

When a unit produces erroneous results, its internal states are likely to be erroneous. Therefore, without recovering a failed unit from the effects of the error, the unit is prone to produce erroneous results in subsequent executions.

So far, the recovery technique which detects errors through the comparison of internal states of multiple units and exchanges a failed unit's internal states for consensus ones has been proposed [5]. This technique imposes restrictions that internal states of units must be identical. The MLDD architecture, placing no constraints on internal states, implements recovery from errors due to design faults of operating system components and hardware components.

The MLDD architecture recovers a failed unit by exchanging the unit's operating system component and hardware component for other components and replaying the execution from the latest checkpoint in which the unit's internal state is error-free. When a faulty component is identified, a failed unit can be recovered by exchanging only the component for another one.

Figure 3 illustrates an example of fault recovery, in which application program A1 triggers design faults of operating system component O1. Unit 1 is recovered by replaying the execution from the latest checkpoint, exchanging operating system component O1 for O2 and hardware component H1 for H2, unless A1 triggers design faults of O2 and H2.

4.2 Fault Identification

In many systems which adopt the design diversity approach, in order to remove design faults revealed in the operational phase, not only the faulty component but also the other components of the unit containing the faulty one are removed from service. During maintenance, a system consisting of N units degrades to a system consisting of N-1 units.

In the MLDD architecture, since any components of a system layer can execute with any components...
of the other system layers, only a failed component is taken off line. During maintenance, a failed component is replaced with another component, and system operation is not degraded while decreasing diversity in the system layer. Since the removal of design faults of VLSI circuits is a time-consuming process, this is very effective in improving system reliability.

In order to provide system maintenance at the component level, identification of a faulty component is required. We will indicate how to identify a faulty component using the replay technique. Let us suppose that a unit fails due to design faults of one component. In order to identify a faulty component, the unit replaces one of its three components with a different one and replays the execution from the latest checkpoint in which the unit's internal state is error-free. If the results of the retry with the faulty unit's hardware component exchanged for a different one are correct, the hardware component is faulty. If only the results of the retry in the case of replacing the faulty unit's application program component with a different one are correct, the application program component contains design faults.

Figure 4 shows that Unit 1 fails due to design faults of operating system component \( O_1 \). In this example, Unit 3 succeeds using operating system component \( O_2 \), and Unit 4 fails, replacing only Unit 1’s hardware component with a different one. Unit 2’s success depends on whether its application program component \( A_2 \) triggers design faults of operating system component \( O_1 \).

5 Examination of Coincident Errors

The MLDD architecture requires a low probability of coincident errors for its improvement of system reliability. Let us suppose that the proposed fault recovery mechanism recovers a unit which fails due to design faults of its operating system component. If the coincident error of the two operating system components occurs, the failed unit is not recovered.

Because coincident errors occur mainly due to unknown factors such as specification ambiguities and common thinking traps of developers, the assurance of a low coincident error rate is difficult. In order to reduce coincident errors, the forced diversity method has been proposed [12]. This method develops diverse implementations through forcing developers to use different developers, specification languages, programming languages, algorithms, data structures, tools, testing techniques and so forth. The PODS project [13] indicated that significantly low coincident error rate was achieved by using different development teams and different combinations of specification languages, programming languages, and algorithms. However, Avizienis and Kelly [14] showed that a positive impact of using different specification languages is too low to justify the additional effort required by them.

We conjecture that the difference between the quality of the development processes leads to the difference between the coincident error rates. In the PODS project, much efforts to avoid design faults were made, but on the other hand, in the experiment conducted by Avizienis and Kelly, the removal of design faults were not encouraged. As a result, the average failure probability of a single program in the PODS project was 0.00000045, when tested with 665,288 random test data, and that of the Avizienis and Kelly experiment was 0.27 in the 100 tests. We conjecture that for a low coincident error rate, high quality development proce-
dures are required.
Moreover, since types of residual faults are closely related to the testing method and test cases, we conjecture that the application of the forced diversity method to the testing phase is effective in the reduction of coincident errors.

The purpose of our examination is to provide some insight into the validity of these conjectures.

5.1 Description of Examination

We examine the faults discovered near the operational phase and in the operational phase of realtime operating systems based on the ITRON2 specification and VLSI CPU chips based on the TRON CPU specification in cooperation with several manufacturers. Our examination is the only case in which implementations are developed with no constraints imposed on the development procedures, using a sufficiently high quality development methodology for real-world applications. The results of the examination will contribute much to the progress of the research on design diversity.

In the rest of this paper, we refer a realtime operating systems based on the ITRON2 specification and a VLSI CPU chip based on the TRON CPU specification as an ITRON2 OS and a TRON CPU chip respectively.

In this examination, we asked several manufacturers about the ITRON2 OS and TRON CPU chip that they have implemented. The questions are as followed.

Question 1 (Verification Technique): In the verification phase,

1. In what development environment do you test your ITRON2 OS (or TRON CPU chip)?
2. What testing techniques do you use?
3. What and how many test cases do you use?
4. What criterion of reliability does your ITRON2 OS (or TRON CPU chip) meet when it is released?

Question 2 (Design Fault): With respect to each error which occurred in the verification phase and in the operational phase,

1. When did the error happen? or, How long have been since your ITRON2 OS (or TRON CPU chip) was released?
2. What system call (or instruction) had the cause of the error?
3. What was the cause of the error?
4. What was the effect of the error?

For estimating a coincident error rate, we will examine whether the design faults of an operating system (or a cpu chip) are contained in the other ones. In addition, we will investigate on the effects of using one specification and different testing techniques on a coincident error rate. For this purpose, we put Question 1 and Question 2-3.

Moreover, the results of this examination are useful in implementing the fault recovery function and the fault identification function, as mentioned in Section 4, repeating execution from the latest checkpoint where internal states of a failed unit are correct. In order to ensure such a checkpoint, the questions, what kind of information should be recorded? and when the information should be saved?, must be clarified. For this purpose, Question 2-3 and Question 2-4 were made.

6 Conclusions

This paper describes the MLDD architecture providing fault-tolerance to design faults in application programs, operating systems, and hardware components. The MLDD architecture applies the design diversity approach to these three system layers with the specifications used for developing components of each layer.

Although the application of the design diversity approach to multiple system layers is a quite powerful approach to ultra reliable systems, little research of it has carried out due to economic reasons. In the TRON project, however, in order to achieve compatibility among systems, independent manufacturers have independently developed several operating systems and CPU chips from the same specifications. This situation resolves the issue related to the cost of introducing design diversity into multiple system layers.

The main feature of the MLDD architecture is compatibility among components of each layer. This compatibility improves system reliability. This paper illustrates that the MLDD architecture implements recovery from design errors of a operating system component and a hardware component and identification of a faulty layer.

In order that the MLDD architecture is effective in system reliability improvement, a low probability of coincident errors is required. We conjecture that a low coincident error rate is achieved by using high quality development procedures and by applying the
forced diversity method to the testing phase. This paper describes the ongoing examination of this conjecture. The examination is also of use for implementing the replay mechanism used in fault recovery and fault identification.

For improving the effectiveness of the MLDD architecture, we will investigate on the development process for diverse failure behavior implementations, applying the forced diversity technique only to the testing phase.

References


