Design Patterns for Safety-Critical Embedded Systems

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften genehmigte Dissertation

vorgelegt von

Ashraf Armoush
aus
Awarta-Palästina

Berichter: Professor Dr.-Ing. Stefan Kowalewski
Professor Dr. Bernhard Rumpe

Tag der mündlichen Prüfung: 15.06.2010

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

Ashraf Armoush
Lehrstuhl Informatik 11
armoush@embedded.rwth-aachen.de

Abstract

Over the last few years, embedded systems have been increasingly used in safety-critical applications where failure can have serious consequences. The design of these systems is a complex process, which is requiring the integration of common design methods both in hardware and software to fulfill functional and non-functional requirements for these safety-critical applications.

Design patterns, which give abstract solutions to commonly recurring design problems, have been widely used in the software and hardware domain. In this thesis, the concept of design patterns is adopted in the design of safety-critical embedded system. A catalog of design patterns was constructed to support the design of safety-critical embedded systems. This catalog includes a set of hardware and software design patterns which cover common design problems such as handling of random and systematic faults, safety monitoring, and sequence control. Furthermore, the catalog provides a decision support component that supports the decision process of choosing a suitable pattern for a particular problem based on the available resources and the requirements of the applicable patterns.

As non-functional requirements are an important aspect in the design of safety-critical embedded systems, this work focuses on the integration of implications on non-functional properties in the existing design pattern concept. A pattern representation is proposed for safety-critical embedded application design methods by including fields for the implications and side effects of the represented design pattern on the non-functional requirements of the systems. The considered requirements include safety, reliability, modifiability, cost, and execution time.

Safety and reliability represent the main non-functional requirements that should be provided in the design of safety-critical applications. Thus, reliability and safety assessment methods are proposed to show the relative safety and reliability improvement which can be achieved when using the design patterns under consideration. Moreover, a *Monte Carlo* based simulation method is used to illustrate the proposed assessment method which allows comparing different design patterns with respect to their impact on safety and reliability.

Acknowledgments

I would like to express my sincere gratitude to my advisor, Prof. Dr.-Ing. Stefan Kowalewski for his considerable guidance, helpful advices, valuable information, and constant support during the development of this thesis.

I would also like to thank Prof. Dr. Bernhard Rumpe for accepting to be a co-referee of my doctorate thesis and his interest in my work. My thanks also go to the committee members, Prof. Dr. Peter Rossmanith, Prof. Dr. Ulrike Meyer, and Prof. Dr. Berthold Vöcking for their time and effort.

I would like to acknowledge the German Academic Exchange Service (DAAD) for granting me the PhD scholarship.

I would also like to express my deepest thanks to Dr. Falk Salewski for his suggestions, useful hints, and helpful advices during the first phase of my work.

Many thanks go to all my colleagues at the Embedded Software Laboratory at RWTH Aachen University, in particular John. F. Schommer, for providing a fruitful working atmosphere. Furthermore, I thank my student Alexander Grinin for helping me to implement the catalog program.

Last but not least, I am thankful and very appreciative to my wonderful wife and my family for their love, encouragement and support during the entire period of my study.

Ashraf Armoush
Aachen, June 2010

Contents

List of Tables

List of Figures

List of Abbreviations

3-LSM	3-Level Safety Monitoring
ADC	Analog to Digital Converter
ASIC	Application-Specific Integrated Circuits
ASIL	Automotive Safety Integrity Level
AT	Acceptance Test
AV	Acceptance Voting
BIT	Built-In Test
CPLD	Complex Programmable Logical Devices
CRC	Cyclic Redundancy Check
CV	Consensus Voting
DSP	Digital Signal Processor
ERM	Entity-Relationship Model
FN	False Negative
FP	False Positive
FPGA	Field Programmable Gate Array
FR	Functional Requirement
GUI	Graphical User Interface
HmD	Homogeneous Duplex
HtD	Heterogeneous Duplex
HW	Hardware
IEC	International Electrotechnical Commission
M-oo-N	M-Out-Of-N
MA	Monitor-Actuator
MISRA	Motor Industry Software Reliability Association
MLV	Maximum Likelihood Voting
MTBF	Mean-Time Between Failures
MTTUF	Mean Time to Unsafe Failure
NFR	Non-Functional Requirement
NSCP	N-Self Checking Programming
NVP	N-Version Programming
OOPSLA	Object-Oriented Programming Systems, Languages, and Applications
P_{SF}	Probability of Safe Failure
P_{UF}	Probability of Unsafe Failure

PLoP	Pattern Languages of Programs
POST	Power-On Self Test
PSC	Protected Single Channel
PV	Plurality Voting
R	Reliability
RB	Recovery Block
RBBV	Recovery Block with Backup Voting
RRI	Relative Reliability Improvement
RSI	Relative Safety Improvement
S_{SS}	Steady-State Safety
SC	Sanity Check
SE	Safety Executive
SIG	Soft-Goal Interdependency Graph
SIL	Safety Integrity Level
SW	Software
TMR	Triple Modular Redundancy
TN	True Negative
TP	True Positive
WD	Watchdog

1. Introduction

Embedded systems are frequently used in a wide variety of applications to realize the control part. These applications range from small applications like mobile phones, or microwave ovens, to complex and critical applications like railways, aerospace, or automotive systems. The common characteristic of embedded systems is that they contain combinations of hardware and software that are built into other systems to perform dedicated functions.

Embedded systems often have special functional and non-functional requirements somewhat different than those required for general-purpose computers. Thus, the hardware and software must be carefully designed with special design techniques to make sure that the final design satisfies the functional and non-functional requirements.

Many embedded systems are described as "*safety-critical*" due to the nature of these applications which include considerable consequences of failures. Failures in such systems could result in critical situations that may lead to serious injury, loss of life, or unacceptable damage to the environment. Therefore, safety and reliability often are more important issues in these applications than performance.

The importance of embedded systems in safety-critical applications has been growing continuously in the recent years. This increasing use introduces the demand for successful, reusable, proven and tested designs that could speed up the development process of new systems.

The concept of design patterns is a universal approach to describe common solutions to widely recurring design problems. A design pattern is an abstract representation for how to solve a general design problem which occurs over and over in many applications. Describing proven solutions as design patterns provides a good documentation for these solutions and makes them more accessible for future use in new systems.

Since design patterns aim to support and help designers and system architects choose suitable solutions for design problems, this concept might also be applicable and useful to support the design of safety-critical embedded systems. The adoption of this concept in the field of safety-critical systems must take into consideration the specific functional and the non-functional requirements of these applications.

While the current concept of design patterns has been addressed for many different domains like software and hardware design, it lacks a consideration of potential side effects on non-functional requirements like safety, reliability, modifiability, cost, and execution time that play a very important role in safety-critical applications. Consequently, the side effects on the main non-functional requirements must be integrated into the current concept of design patterns to transfer this concept to the field of safety-critical embedded systems.

Due to the missing representation of non-functional side effects and implications in the currently used pattern templates, a new template representation should be used to construct a collection of design patterns for safety-critical embedded systems. Most of the presented patterns in this thesis are not new. They describe well known and proven design methods that have been applied many times to solve specific problems in the development of safety-critical embedded applications. However, they are presented and analyzed based on the considerations described above.

1.1. Objectives

The main objective of this thesis is to construct a catalog of design patterns which helps designers to build safe embedded systems for safety-critical applications with the help of a structured, effective, simple, and quickly accessible collection of successful solutions.

Widely used and proven solutions in the field of embedded-system have to be collected from literature to be used in this collection. These solutions should be generalized and captured in a form of design patterns to fit to similar design problems in the field of safety-critical embedded systems.

Furthermore, the non-functional requirements for these patterns should be studied and classified to achieve the following goals:

- Establishing a high level and abstract representation which can be used to document the impact of the collected solutions as design patterns on these requirements.
- Constructing a reliability and safety assessment method which can be employed to facilitate the comparison possess of design patterns for their suitability for a specific design problem.
- Developing a systematic search method for decision support to give designers an automated recommendations support of suitable design patterns for a given application. These recommendations are based on the requirements of design patterns and the available resources for a desired application.

1.2. Contributions

The main contributions of this thesis are as follows:

- We have proposed a new pattern template for safety-critical design patterns. It includes the classical representation in addition to fields for implications and side effects on non-functional requirements. The considered requirements include safety, reliability, modifiability, cost, and execution time.
- We have collected fourteen proven design techniques for hardware and software. We have then analyzed, studied, and represented these techniques using the new template.
- We have developed a new software pattern called *Recovery Block with Backup Voting (RBBV)* to improve the reliability of the classical *Recovery Block (RB)* method in those situations where it is difficult to construct an effective acceptance test.
- We have developed a new reliability metric for design patterns which describes the reliability improvement relatively to a basic system.
- To provide a safety assessment method at the abstract level of design patterns, we have developed a method which consists of two parts:
    1. A system of safety recommendations for safety-critical design patterns. These recommendations are based on the safety recommendations of the *IEC 61508* [46] standard, and contain three additional types of recommendations.
    2. A safety metric based on the computation of the relative safety improvement achieved when using the design patterns under consideration.
- In order to illustrate the proposed reliability and safety assessment method, we have conducted a test case that includes four software design patterns sharing the concept of software diversity. In this case, we have developed a *Monte Carlo* based simulator to demonstrate the proposed safety and reliability metric.
- To provide a decision support for users in the intended catalog, we have categorized the possible applications into six specific problems in addition to a general design problem. Moreover, we have classified the possible available resources and application requirements to be used as guidelines to find applicable solutions for each design problem.

- We have constructed a data model to store the design patterns and their related information. We have then implemented the catalog of design pattern which includes documentation for fifteen design patterns in addition to a user decision support as a desktop application.

1.3. Thesis Structure

The rest of this thesis is organized as follows: Chapter 2 gives a description of preliminaries of this research, including embedded systems, safety-critical systems, functional and non-functional requirements, and the concept of design patterns. Chapter 3 presents the classical representations for design patterns and the new proposed template.

Chapter 4 describes safety and reliability assessment method proposed for safety critical design patterns. Chapter 5 presents a case study, which demonstrates the use of the developed assessment method. It also contains the results of the conducted simulation for four design patterns.

Chapter 6 describes the structure and basic features of the develop catalog program. The main part of this thesis, which contains the documented design patterns, is presented in Chapter 7 to 9.

Finally, Chapter 10 summarizes this thesis and outlines possible issues for future work.

1.4. Bibliographic Notes

Some parts of this thesis are based on work that has been previously presented in earlier publications. The new pattern representation template with a comparison to currently used classical representations was published in [6, 9]. The effect of a weak acceptance test on *Recovery Block* method and the basic idea of the *Recovery Block with Backup Voting* as a fault-tolerant method was published in [7]. Later, the *Recovery Block with Backup Voting* was published as design pattern in [8]. Regarding the developed safety assessment method for safety-critical design patterns, the safety metric was published in [4] while the system of safety recommendations was published in [5].

2. Preliminaries

This chapter provides some preliminaries for this thesis. The first two sections introduce embedded and safety-critical systems. Section 2.4 presents the basic concepts of design patterns, while Section 2.5 offers a brief literature review of the use of design patterns.

2.1. Embedded Systems

Nowadays, embedded systems are widely used in various applications and devices such as mobile phones, microwave ovens, cars, medical devices, aircrafts, telecommunications, trains, and power plants. The growing importance of embedded systems in many fields is largely driven by advances in microelectronics and software.

Unlike general-purpose computers, embedded systems are dedicated to perform application-specific functions. Such systems are designed to interact with a larger embedding system, where they should be enclosed or embedded, to satisfy the requirements for an application. Usually, an embedded system includes sensors and dedicated input devices to collect information about the surrounding environment, and actuators to control that environment.

While embedded systems are widely associated with microprocessors and microcontrollers, embedded systems can contain a variety of other technologies such as digital signal processors (DSPs), complex programmable logical devices (CPLDs), application-specific integrated circuits (ASICs), and field programmable gate arrays (FPGAs). In embedded systems, the developed software is often dependent on the used hardware, and the design of the hardware and software influence each other. Therefore, embedded hardware and software must be concurrently designed or the interface must be clearly defined [64, 99].

Based on these aspects, there are many definitions for embedded systems (see e.g. [99, 18, 74]) but one of the most general definitions is: "*Embedded systems are information processing systems that are embedded into a larger enclosing product*" [71].

In the most general case, an embedded system consists of a combination of hardware and software components that interact with the physical environment through sensors and actuators to perform a dedicated task. Due to the tight relation between hardware and software in embedded systems, they should be carefully designed to make sure that implementation meet the main requirements for the embedded system and also for the enclosing product.

2.2. Characteristics of Embedded Systems

Despite the variety of embedded systems applications and implementation methods, there are some common characteristics in these systems that can be summarized as follows:

Dedicated Functionality

Since an embedded system is normally used to perform the control and information processing for the enclosing system, it accomplishes some predefined tasks that are related to the given application. This is in contrast with general-purpose computers where it is more typical to run a large variety of applications.

Limited Resources

For many reasons (like nature of applications, marketing issues, production costs, or hardware technology), most of today's embedded systems face stringent constraints in terms of limited resources such as memory size, hardware functionality, input and output resources, clock frequency, power consumption, operating system, weight, size, and so forth.

Performance and Efficiency

Achieving high performance, which is most of the time characterized by the amount of work done and the execution time, is of great importance in many embedded systems. Due to the constraints of limited resources, embedded systems have to be also efficient in many aspects such as memory usage, power consumption, and hardware resources. In general, improved performance represents a challenging optimization problem that involves several contradictory requirements. Therefore, designers of embedded systems must clearly understand the application and its associated constraints [77].

Real Time Constraints

Time in embedded systems is often one of the acute constraints that are not found in typical computer systems. Most embedded systems have to react to external events and perform computational tasks within precise timing constraints determined by the environment. These systems, where the correctness depends on the output results as well as on the time at which the results are produced, are classified as "Real-Time Systems". (see e.g. [21, 24] for more classifications and trends in real-time embedded systems).

Interaction with Environments

Commonly, embedded systems provide control and sensing to the enclosing systems, thus they interact continuously with environments through sensors, transducers, and actuators. In many cases, the interactions between an embedded system and its environment are hidden from the user who just uses dedicated input devices or limited graphical user interfaces.

Dependability

Dependability[1] is defined in [14, 15] as: "*the ability of system to avoid failures that are more frequent or more severe, and outage durations that are longer, than is acceptable to the user(s)*". Therefore, the term dependability as a system property is highly related to the safety-critical applications or to the applications that involve direct impact on the environment or on the users. As more and more embedded systems are deployed in our daily life, designers should ensure that these systems are dependable.

The notion of dependability as presented in [15] covers a list of attributes which includes:

- ***Availability***: readiness for correct service.
- ***Reliability***: continuity of correct service.
- ***Safety***: absence of catastrophic consequences on the user(s) and the environment.
- ***Integrity***: absence of improper system alterations.
- ***Maintainability***: ability to undergo modifications and repairs.

In this thesis, we will mainly focus on the safety and reliability as major non-functional requirements for safety-critical embedded system, which does not imply that the other attributes are less important. Further information about the other attributes can be found in [62, 97, 14, 15].

[1]The various dependability concepts, which are used throughout this thesis to describe aspects of safety-critical systems, like dependability, reliability, safety, availability, fault, failure, error, fault detection, fault masking, fault tolerance, fault removal, and fault prevention, are compliant with the taxonomy and basic concepts presented in [15].

2.3. Safety-Critical Systems

The definition of *safety-critical system* can be derived from the definition of *safety* which is: "*a property of a system that it will not endanger human life or the environment*" [97]. Therefore, the "*Safety-Critical Systems*" is used to describe those systems or applications in which failure can lead to serious injury, loss of life, significant property damage, or damage to the environment [54, 38, 39].

Based on this definition, there are many fields that can be classified as safety-critical such as medical care devices, nuclear plants, railways, weapons, automotive industry, and aircrafts. The design of these systems is considered to be a complex process, which involves the integration of well-known design strategies and techniques in hardware and software to fulfill two types of requirements:

- **Functional Requirements (*FRs*)**: the functions to be performed, or what an embedded system is expected to do.
- **Non-Functional Requirements (*NFRs*)**: the quality attributes of a system as it performs its job [90]. Moreover, they include evolution qualities, which have to be provided in a system, such as modifiability and cost.

The functional requirements are clearly important for any computer system, but in the field of safety-critical systems, the non-functional requirements are also of great importance to be provided in the final system. The non-functional requirements for safety-critical systems include a set of requirements like high reliability, safety, availability, ease maintainability, low cost, and small size. While the relative impotence of these requirements varies from one application to another, all safety-critical systems must guarantee an adequate level of safety.

Ensuring safety in the design of a safety-critical embedded involves a lot of analysis and testing. Since safety property is determined by the possible failures that can lead to a hazardous situation, one of the mostly used techniques is hazard[2] analysis which is at the heart of any safety-critical system [66]. Hazard analysis is an iterative method that is carried out throughout all the development phases with purpose to: identify the unsafe states, evaluate the risk of the hazards, and to identify the necessary measures that can be taken to deal with these hazards [47].

System safety consists of two parts, hardware safety and software safety. Different techniques are normally used to reduce hazards and enhance safety in the two parts. In hardware, redundancy and diversity are the most common

[2]Hazard: "*is a situation in which there is actual or potential danger to people or to the environment*" [97].

techniques, while in software the techniques includes design diversity, hazards prevention, hazard detection, or controlling hazards when they occur [67]. Moreover, designers of software and hardware for a safety-critical embedded system must ensure the safe integration of the two parts to guarantee the safe operation of this system.

Generally, failures in safety-critical systems are the results of errors which are in turn the results of faults. Therefore, the general techniques that are used to successfully attain dependability (which includes safety) can be also used to enhance system safety. These techniques can be categorized according to the stage in which the technique is performed (see e.g. [81, 15]):

- ***Fault Avoidance or Prevention****: how to avoid or prevent occurrence of faults.* These techniques are conducted during system development to reduce the number of introduced faults.

- ***Fault Removal****: how to reduce the number of faults.* This is the second step, and includes testing and inspection.

- ***Fault Tolerance****: how to prevent system failures from occurring.* The fault tolerance techniques are employed during the development time to enable the system to tolerate remaining faults and to deliver correct service in the presence of faults.

- ***Fault Forecasting****: how to provide system evaluation, via estimating the present number, the future incident and consequences of faults.* These techniques are used to assess the fault tolerance robustness.

It is important to note that fault prevention, removal and tolerance should not be regarded as alternatives [83], but rather they should be considered as complementary techniques. The degree of success of an effective combination of these techniques can be evaluated with fault forecasting techniques

Finally, safety considerations must be taken into account throughout the entire development process of safety-critical embedded systems. Furthermore, it is necessary to provide an appropriate and unified system of judgment to reflect the importance of safe and correct operations in the target applications. Thus, many safety standards have been developed to cover the safety management of safety-critical systems throughout their lifecycles. Some of the important and commonly used standards are: *MIL-STD-882D* [35] which is a military standard and *IEC61508* [46] which is a well-known application-independent standard. Further information about safety standards will be presented in Chapter 4.

2.4. Concept of Design Patterns

The concept of design patterns became one of the widely used and universal approaches for describing and documenting recurring solutions for design problems. The idea of design patterns was original proposed by the architect *Christopher Alexander* [2] who wrote several books on the field of urban planning and building construction. He defined a design pattern as a construct that expresses a relation between three parts: *context, problem* and *solution*(what is called *three-part rule*). Ever since, this concept has been applied to many different domains including hardware and software design.

In software domain, design patterns became popular in software architecture and development after the success of the book *Design Patterns: Element of Reusable Object-Oriented Software* by *Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides* [41] frequently referred to as the *Gang of Four (**GoF**)*. They defined a design pattern as *Description of communicating objects and classes that are customized to solve a general design problem in a particular context.* This book, which includes a collection of patterns for object-oriented software design, was not the first effort to use the concept of design pattern in software; however, it is considered as the foundation for design patterns in software construction.

Like in software development, design patterns have been also adapted for hardware design to provide implementation-independent and abstract views for recurring hardware design solutions. The more general benefit of using design pattern is to improve the hardware development process through importing the advantages of object oriented modeling techniques (see e.g. [31, 32, 88, 102]).

In general, we can summarize a design pattern as an abstract representation for how to solve a general design problem that occurs over and over in many applications. The main purpose of design patterns is to support and help designers and system architects choose a suitable solution for a recurring design problem among available collection of successful solutions. Moreover, design patterns can simplify communication between practitioners and provide a good way for documentation of proven design techniques.

2.5. A Brief History of Design Patterns

While the *GoF* book [41] has been the most popular work on design patterns over the last two decades, there have been several attempts in the literature to adapt this concept in many fields of system design. In 1987 *Cunningham and Beck* presented five patterns for designing windows-based user interfaces with Smalltalk [16]. Around the same time, *Jim Coplien* began to document *C++*

Idioms that represent specific constructs like patterns for *C++*. He published these idioms as a book in 1991 [29], and later he published these idioms and patterns as paper in 1997 [30].

From 1990 to 1993, several pattern papers, which addressed the use of design pattern in object oriented programming, were published at the *OOPSLA (Object-Oriented Programming Systems, Languages, and Applications)* conference. In 1994, the *Hillside group* (*see http://hillside.net*) organized the first *PLoP (Pattern Languages of Programs)* annual conference. The revised papers from *PLoP* are normally published in the book series "*Pattern Languages of Program Design*" (see e.g. [69]). Meanwhile, *Buschmann et al.* published the book "*Pattern-oriented software architecture: a system of patterns*" [23] which includes a collection of relatively independent solutions to common design problems represented as a catalog of design patterns.

Another well known work is the catalog presented by *Bruce Douglass* in [37]. This catalog includes a set of patterns for real-time embedded systems. The presented patterns deal with real-time design issues like concurrency, resource sharing, distribution, and safe and reliable architecture.

Finally, the concept of design pattern has become an important area of research in many fields like: fault tolerance [45], telecommunications [89], embedded systems [78, 79], security [94, 103], and many other fields. Each of these fields has its own patterns and sometime its own representation, but all follow the basic principle of design patterns.

3. Pattern Representations

This chapter describes the advantages of including the side effects on non-functional requirements in the proposed pattern representation. The first section presents parts of the traditional pattern representation that are used in all patterns. After that, we give an overview about other commonly used representations. Section 3.3 gives the motivation for using a new template representation. Next, we review the related work to our approach for safety-critical design patterns and the non-functional requirements in Section 3.5. The last section presents the template which is used in our catalog to represent design patterns.

We published the general idea of the new template representation in two papers [6, 9].

3.1. Traditional Pattern Representation

The pattern representation as proposed by *Alexander* was derived from the definition of design patterns. He defined a design pattern as a construct that includes a relation between three parts: context, problem, and solution. This three-part rule indicates the general structure that should be provided in any pattern representation form.

In generally, the traditional pattern representation consists of four essential elements:

- ***Name***: a meaningful name.
- ***Context***: describes the preconditions or the general situation in which the pattern can be used to solve the problem.
- ***Problem***: describes the problem which is addressed by the pattern.
- ***Solution***: explains a solution to the problem under consideration.

This structure shows the basic relation between the main parts: the problem to be solved, the appropriate context that should be satisfied in order to apply the pattern, and the solution presented by the pattern. Moreover, the solution must be abstract and independent from specific implementation, which is one of the main attractive issues in design patterns.

3.2. Other Representations

Based on *Alexander's* traditional form of representation presented in the previous section, several representation forms have been proposed and used to give a highly structured and abstract description for design patterns. Tab. 3.1 shows some of the alternative representations defined by different authors for the description of design patterns.

While each form has been used to provide a uniform way of documentation and communication between experts and practitioners in a specific field, there is no uniform agreement between these forms on a single template. Some representation forms include precise and short structure, while others include more comprehensive and expressive forms. In some cases, the authors chose different names to describe the same element: e.g. (*Context, Applicability, and Preconditions*) and (*Intent, Motivation, and Purpose*). Despite of diverse formats, names and domains, these representations should at least contain the essential elements as defined by the traditional representation.

3.3. Motivations for a New Representation

As shown in Section 2.3, the non-functional requirements, which include the quality attributes of the system as it performs its tasks, are of great importance in safety-critical applications and should be considered during the development process of such systems. Furthermore, these non-functional requirements should be represented and taken into consideration in the high-level representations of design methods such as in design patterns.

While the concept of design patterns has been extensively studied and used in various fields of system design, it typically lacks a consideration of potential consequences and side effects on non-functional requirements. Some of the well-known representations (e.g. *GoF*) initialize a field for consequences, but most of the time this field deals with the impact of a design pattern on space, time, system's flexibility, system's portability, and other implementation issues [41]. Nevertheless, none of the used forms includes the impact of a design pattern on the non-functional requirement (safety, reliability, modifiability, cost, and execution time) in its representation.

The missed representation of the implication on these non-functional requirements in the currently used templates reveals the need for new template that includes a part for these requirements. This template will be used to achieve the main objective of this thesis which includes the construction of a catalog of design patterns for safety-critical embedded systems.

Table 3.1.: Alternative pattern representations [51].

	GoF[41]	**Züllighoven and Riehle** [87]	**Adams** [1]	**Rubel** [93]	**Lajoie and Keller** [58]
Name	✓	✓	✓	✓	✓
Known as	✓				
Aliases			✓		
Intent	✓				✓
Purpose		✓			
Context		✓		✓	
Preconditions			✓		
Applicability	✓				✓
Problem		✓	✓	✓	
Solution		✓	✓	✓	
Description					✓
History			✓		
Diagram					✓
Design Rationale				✓	
Motivation	✓				✓
Structure	✓				
Participants	✓				
Collaborations	✓				
Consequences	✓				
Implementation	✓				✓
Sample Code	✓				
Known Uses	✓				
Related Patterns	✓			✓	
See also					✓
Compare		✓			
Constraints			✓		
Forces				✓	
Category					✓
Discussion					✓
Contract Example					✓
Resulting Context				✓	

3.4. Design Pattern Template

This section presents the new template which will be used to provide a consistent representation for design patterns in our catalog.

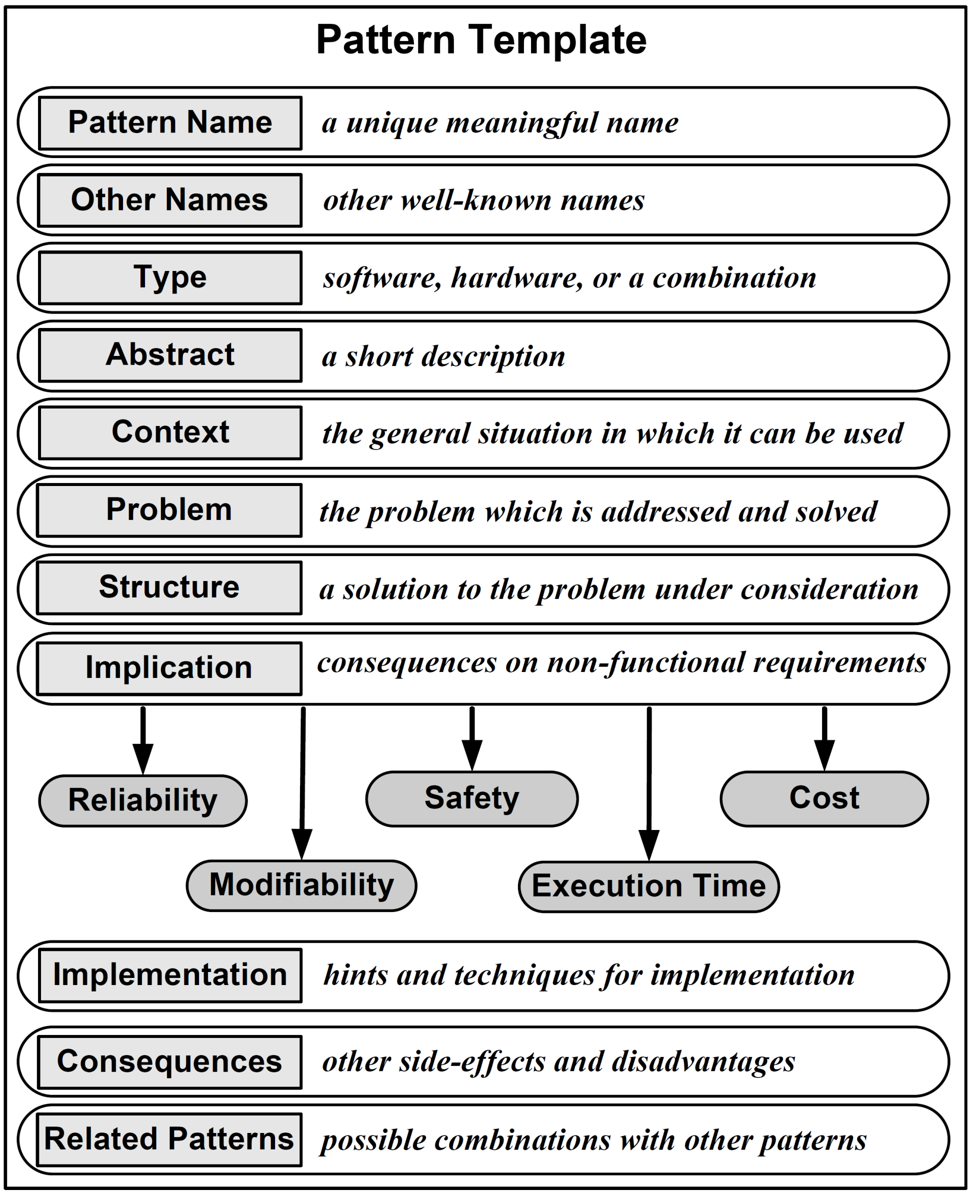

Pattern Template
Pattern Name
a unique meaningful name
Other Names
other well-known names
Type
software, hardware, or a combination
Abstract
a short description
Context
the general situation in which it can be used
Problem
the problem which is addressed and solved
Structure
a solution to the problem under consideration
Implication
consequences on non-functional requirements
Reliability
Modifiability
Safety
Execution Time
Cost
Implementation
hints and techniques for implementation
Consequences
other side-effects and disadvantages
Related Patterns
possible combinations with other patterns

Figure 3.1.: Design pattern template.

As depicted in Fig. 3.1, the upper part of this template includes the traditional representations of design patterns while a listing of the pattern implications on non-functional requirements is given in the *Implication* section. Moreover, further support is given by stating implementation issues, summarizing other consequences and side effects as well as a listing of related patterns.

The pattern template consists of the following elements:

- **Pattern Name**
  A meaningful name (*list of words*) to describe the pattern uniquely. This name will be used as a handle to this pattern.
- **Other Names**
  The other well-known names for the pattern, if exist.
- **Type**
  Gives the classification of the design pattern into:
  - *Hardware*: when the pattern contains hardware redundancy dedicated to handle hardware fault.
  - *Software*: when the pattern contains software diversity to tolerate software faults.
  - *Combination of hardware and software*: other cases.
- **Abstract**
  A short description for the design pattern.
- **Context**
  The general situation in which the design pattern can be applied.
- **Problem**
  This part gives a summary of the problem which is addressed and solved by this pattern.
- **Structure**
  This is the main part of the pattern, since it represents a solution to the problem under consideration. It provides the main elements of the pattern, the relation between these elements, and how they cooperate to solve the problem.
- **Implication**
  The consequences on non-functional requirements. This part is divided into five sections, where each one will deal with one of the following:
  - ***Reliability***: gives the relative improvement in the system's reliability achieved by this pattern.

  - ***Safety***: gives the safety recommendations for the pattern in addition to the relative safety improvement that can be achieved.
  - ***Cost***: the implications on costs include:
    1. The recurring cost per unit, which reflects the additional costs resulting from additional or specific hardware components required by the design pattern.
    2. The development cost for this pattern.
  - ***Modifiability***: describes the degree to which a system developed according to this design pattern can be modified and changed.
  - ***Impact on execution time***: with this implication, the effect of the pattern on the total time of execution at runtime is indicated. It shows the execution time overhead that is resulting from the application of this pattern in the worst and average cases.

- **Implementation**
  This part gives the aspects, hints, and techniques that should be taken into consideration when implementing the pattern.

- **Consequences**
  Includes side-effects and disadvantages that may appear due to the application of this pattern.

- **Related Patterns**
  The closely related design patterns to this pattern, and the possibility to combine the related patterns with this one.

Chapter 4 gives more details about reliability and safety assessment methods which have been used in this thesis to show the implications of the design pattern on safety and reliability.

3.5. Related Work

As the field of design pattern software is rapidly growing, many research have focused on the use of design pattern in software domain [41, 23, 28, 16, 30]. Nevertheless, further research is still needed in the domain of safety-critical embedded systems to integrate the non-functional requirements (basically safety) in design patterns.

This section presents related work regarding design patterns for safety-critical systems and the integration of non-functional requirements in design patterns.

3.5.1. Patterns for Safety-Critical Systems

In the field of safety-critical patterns, the most popular work is in [36, 37]. *Bruce Douglass* propose several design patterns for safety-critical systems based on well known fault tolerant design methods and by integrating some modification to increase the safety level on these patterns.

A further work, which gives a way to construct software safety patterns, is in [100]. *Wu and Kelly* present a method for software architecture design for safety-related systems. This method is based upon extending the notation of architectural tactics to include safety as a consideration. The architecture tactics are design decisions that aid designers in selecting appropriate techniques to achieve quality attributes. Furthermore, related tactics can be combined into a pattern that controls a specific quality attribute.

3.5.2. Non-Functional Requirements in Design Patterns

To establish a connection between the non-functional requirements and the concept of design patterns, many research have been conducted in this context. *Gross and Yu* [43] discuss the relationship between non-functional requirements and design patterns, and propose a systematic approach for evaluating design patterns with respect to non-functional requirements. They propose the use of design patterns for establishing traces between non-functional goals in a goal tree such as a soft goal interdependency graph (*SIG*) and the system design. *Cleland-Huang et al.* [27, 40] enhance the patterns defined by *Gross and Yu* in [43] through defining a model for establishing traceability between certain types of non-functional requirements and code artifacts, through the use of design patterns as intermediary objects. *Xu et al.* [101] classify the dependability needs into three types of requirements and proposed an architectural pattern that allows requirements engineers and architects to map dependability requirements into three corresponding types of architectural components.

Bitsch [20] gives a method to use a formal notation to specify the safety requirement with the help of safety patterns. Later, *Konrad et al.* [55, 56] describe a research of how the principle of design pattern can be applied to requirements specifications, which they term requirements patterns for embedded systems. They include a constraints field in the pattern template to show the functional and non-functional restrictions that are applied to the system.

In comparison to our work, none of the aforementioned approaches shows clearly the implications on the non-functional requirements as part of the pattern. These patterns and the later developed patterns focused on the traditional structure of the pattern that includes: context, problem and solution. The use of non-functional requirements in these approaches is restricted to the requirement analysis phase of the design process, or as a constraint field in the template. Moreover, they do not give a relative measure or indication to the implication of the pattern on non-functional requirements.

4. Safety and Reliability Assessment

This chapter presents in detail the implications of design pattern on safety and reliability. The first section describes the idea of computing the relative reliability improvement. Section 4.2 gives a brief overview of safety standards. Section 4.3 shows the importance of safety and risk metrics and gives some examples of common metrics. Section 4.4 describes the basic idea of safety integrity levels as presented in IEC 61508 standard. The last section presents a safety assessment method for design patterns. This method includes a system of safety recommendations for design patterns and a safety metric for the relative safety improvement.

We published the general idea of safety metric for design pattern in [4] and the system of safety recommendations in [5].

4.1. Reliability Assessment

Reliability is conventionally defined as "*a characteristic of an item, expressed by the probability that the item will perform its required function under given conditions for stated time interval*" [19]. Based on this general definition, it represents an essential requirement for safety-critical systems, especially in those situations where it is difficult to reach a *fail-safe state*[1]. For example, in avionic systems there is usually no state that can be considered to be fail-safe. Thus, such systems must continue operating correctly to assure safety.

It is clear from the above definition that reliability of a safety-critical system is totally related to probability of faults that could lead to system failure. Consequently, several fault-tolerance methods and techniques have been proposed and used to tolerate hardware and software faults which, in turn, lead to system reliability improvement.

The best-known measure for reliability is a mathematical measure $R(t)$ which gives the probability that the system is functioning correctly over the time interval $[0, t]$ [57]. The increase in this measure, which is achieved after using a specific design technique, can be used to express the improvement in reliability.

[1]**Fail-Safe State**: a state which, in the event of failure, can be identified as being safe without risk.

4.1.1. Limitations

A design pattern represents a high level abstract solution to a commonly recurring design problem. It is not related to a specific application or to a specific implementation; thus, it is very difficult to find an actual value for probability that the system has been up continuously and without failures after applying the pattern under consideration.

4.1.2. Basic System

The main objective of the implication part in the pattern template is to give an indication about side-effects on non-functional requirement after using the design pattern under consideration. This part will be devoted to make an assessment and compression between different applicable patterns for a specific design problem.

As a result to the previous limitations, instead of giving an actual value for the implication on non-functional requirements, the implication on reliability and safety in this context will be calculated relative to a common basic system.

The basic system which will be used in our approach is shown in Fig. 4.1. It contains a single processing hardware channel with single version software and without specific safety functions.

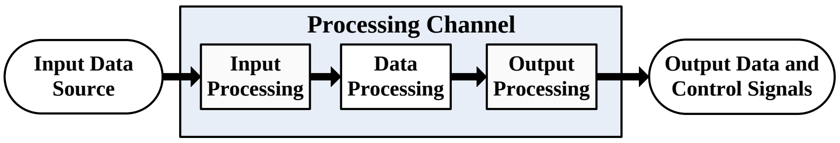


Figure 4.1.: Basic system without specific safety requirements.

4.1.3. Relative Reliability Improvement

As mentioned above, the implication on reliability will be stated relative to the basic system that we have previously defined. We will use a metric called *Relative Reliability Improvement (**RRI**)* which is defined as the percentage improvement in system reliability relative to the maximum possible improvement.

The maximum possible improvement in reliability is the difference between reliability of the basic system and the maximum value for reliability which is equal to 1.0 (*Ideal case without failure*).

$$RRI = \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% \tag{4.1}$$

– *RRI: Relative reliability improvement.*
– R_{old}: *Reliability of the basic system.*
– R_{new}: *Reliability after using the design pattern.*

This metric can be easily applied in the assessment process of design patterns: either through employment of a mathematical modeling or by using simulation techniques similar to the method used in our case study (see Chapter 5).

4.2. Safety Standards

In safety-critical embedded systems, ensuring adequate safety level represents the major factor in the success of these systems. Often, there is no single safety technique that can be applied at a specific point to reach the desired safety requirement. Conversely, special aspects, requirements, techniques, and safety management procedures should be considered in all stages of system development lifecycle.

Consequently, many safety standards have appeared in various domains, such as military, automation, avionic, healthcare and general industry. These standards have played a major role in the development of safety-critical systems by providing a consistent safety management throughout their life-cycle.

Four important safety standards from different fields are discussed briefly in the following sections.

4.2.1. IEC 61508

IEC 61508 [46] is titled "Functional safety of electrical/electronic/programmable electronic safety-related systems". It is a generic industrial safety standard that was approved by the *International Electrotechnical Commission (IEC)* to cover functional safety in all safety-related systems that are electrotechnical in nature irrespective of their application. Therefore, it is intended to be applicable to any industrial safety-related system.

The standard consists of seven parts that cover the complete safety life cycle. The main three parts 1-3 present general functional safety management, hardware aspects and requirements, and requirements for software design of the safety-related systems. Parts 4-7 provide supplementary material that includes definitions, abbreviations, examples, guidelines on the application of software and hardware requirements, and an overview of safety measures and techniques. (see e.g. [96] which gives a guide to applying the IEC 61508 standard)

4.2.2. MIL-STD-882D

MIL-STD-882D [35], which is titled "Standard Practice for System Safety", was approved for use as a military standard by the *US Department of Defense (DoD)* in 2000. It addresses the management of environmental, safety, and health mishap[2] risks encountered in the development, test, production, use, and disposal of *DoD* systems, subsystems, equipment, and facilities. Moreover, it provides the basic safety requirements to perform throughout the life-cycle for any safety-related system as well as guidelines to help user in applying the standard.

4.2.3. ISO 26262

ISO/WD 26262 [48] is titled "Road vehicles - Functional safety". It is the adaptation of IEC 61508 to meet needs specific for automotive sector. The standard includes ten parts to cover all activities during the safety life-cycle of safety-related systems consists of electrical, electronic, and software elements that provide safety-related functions.

This standard uses automotive specific safety integrity levels instead of the safety integrity levels [3] provided in IEC 61508. Similar to the other standards, it provides requirements for confirmation measures to ensure a sufficient and acceptable level of safety is achieved.

4.2.4. DO-178B and DO-254

DO-178B [91] and DO-254 [92] are two correlated standards issued by *RTCA, Inc.* and accepted by the *Federal Aviation Administration* to give guidance on the development of safety-related systems in airborne environments. The former presents guidance that covers the development of software for airborne systems, while the later covers the design of safety-related hardware in these systems.

4.2.5. Choice of standard for our work

Clearly, IEC 61508 is the most general standard that can be applied to any safety-related system; thus, it has been used in our research to derive a safety assessment method for design patterns that present abstract solutions.

[2]**Mishap** is defined in (*MIL-STD-882D*) as an unplanned event or series of events resulting in death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment [35].

[3]See Section 4.4 for the concept of safety integrity levels

4.3. Safety and Risk Metrics

As shown in Section 2.3, system safety is related to the risk of failure in a system and the analysis, techniques and procedures that should be conducted to reduce the risk to an acceptable level. In this context, there is a major concern about the safety and risk assessment of these systems.

In order to provide suitable safety assessment methods for the assessment of safety-critical systems, there is a need for suitable safety and risk metrics. Therefore, many metrics, such as *Steady-State Safety* (S_{SS})[4] and *Mean Time to Unsafe Failure* ($MTTUF$)[5], have been proposed to be used in the assessment of safety-critical systems (see e.g. [34]). Most of these metrics evaluate system safety as a function of time and do not concentrate on the consequences of unsafe failures.

Meanwhile, risk, which is defined in the standard *IEC 61508* as a combination of the probability of occurrence of harm[6] and the severity of that harm, is considered as the most generic metric that deals with a wide range of applications. The risk metric is based on the following equation:

$$R = C \times f \tag{4.2}$$

– R: *the risk in the system.*
– C: *the consequence of the hazardous event.*
– f: *the frequency of the hazardous event.*

The aim of the following sections is to construct an abstract safety metric, similar to the above generic risk metric, to be used in the assessment process of design patterns.

4.4. Safety Integrity Levels

The first part of our safety metric is derived from the safety integrity levels presented in IEC61508 standard; thus, this section gives a brief overview of the concept of safety integrity levels.

The term Safety Integrity as defined by IEC 61508 is "*the likelihood of a safety-related system satisfactorily performing the required safety function under all stated conditions within a stated period of time*". Unfortunately, the implications

[4]**Steady-State Safety**: a probability representing the evaluation of safety as a function of time, in the limiting case as time approaches infinity [104].
[5]**Mean Time to Unsafe Failure**: the expected time that a system will operate before the first unsafe failure occurs [104].
[6]**Harm**: physical injury or damage to the health of people either directly or indirectly as a result of damage to property or to the environment [46].

of failure vary largely between different applications, which arises the need for a method to express the importance of correct and safe operation in a safety-related system.

The idea is to divide the spectrum of integrity into discrete levels, which are called *Safety Integrity levels (SIL)*. A safety integrity level specifies the safety integrity requirements of the safety function to be allocated to the safety-related system.

The concept of safety integrity levels have been adopted by many safety standards. IEC 61508 standard defines discrete levels for specifying the safety integrity requirements of safety functions to be allocated to the system. These levels range from level 1 (*SIL1*) to level 4 (*SIL4*), where safety integrity level 4 denotes the highest level of safety integrity. Like IEC 61508, ISO 2626 defines automotive safety integrity levels (*ASIL*) which include four levels (*A*-*D*). Also, *MISRA*[7] guidelines [73] include five levels that are defined qualitatively rather than mapping integrity to numerical range [96].

Despite the different notations and classifications used for safety integrity levels in safety standards, this concept represents a useful approach to show the importance and implication of correct operation in a safety-related system.

4.5. Safety Assessment

While several assessment methods have been used to evaluate safety-critical systems, most of these methods can not be used to assess safety-critical design patterns due to the abstract nature of these patterns.

The objective of this part is to provide a safety and risk metric that can be used to show the implication of design patterns on safety and also to facilitate the assessment process of design patterns for safety-critical embedded systems. This metric is derived from the general risk metric shown in Equation (4.2).

Unlike real safety-critical systems, design patterns are abstract solutions that are not related to specific applications or implementations. Consequently, it is very difficult to find actual values for both the frequency of hazardous event (f) and the consequences of the hazardous event (C). This limitation reveals the need for new parameters that can be used to reflect the idea of the original parameters, the frequency of hazardous event and the consequences of the hazardous event, in the context of design patterns.

[7]**MISRA**: Motor Industry Software Reliability Association (see http://www.misra.org.uk/)

4.5.1. Applicability to Safety Integrity Levels

The first part of our risk metric is based on the safety integrity levels presented by the IEC61508 standard. While a safety integrity level is derived from an assessment of risk, it is not a measure of risk [86]. The safety integrity levels will be used in the first component of the risk metric as an indication of severity of failures in the considered application after applying a specific design pattern.

The IEC 61508 standard provides a method to show the importance of design methods, techniques, and measures that are applied in order to avoid safety-critical failures caused by hardware and software. This method includes recommendations for these techniques and measures. These recommendations are dependent on safety integrity levels and classified according to their importance into four types [46]:

- **HR**: the technique is highly recommended for this safety integrity level.
- **R** : the technique is recommended for this safety integrity level
- – : the technique has no recommendation for or against used.
- **NR**: the technique is positively not recommended for this safety integrity level.

It is important to note that the use of these recommendations does not guarantee the satisfaction of the required safety integrity in the final design. Instead, the standard should be followed throughout the entire life-cycle in order to get a certificate for a specific safety integrity level.

While the safety recommendations are given in IEC 61508 for different software and hardware techniques, these recommendations can not be directly derived for general design patterns or architectures. Usually, a design pattern presents a combination of different techniques and design methods that cooperate to improve more system properties. Hence, it is necessary to provide a method that can be applied to derive general safety recommendations for safety integrity levels at the abstract level of design patterns.

Table 4.1.: Safety recommendations in IEC 61508.

Recommendations	**Value**
NR	0
–	1
R	2
HR	3

The first step in our approach is to assign equivalent integer values for the different recommendations as shown in Tab. 4.1 The second step is to compute an equivalent integer number for the design pattern under consideration. Since it is difficult to estimate the different effects of the different techniques that constructs a design pattern, it is not possible to use heuristic or different weights for these techniques. Consequently, equivalent weights are assigned to the existing techniques and the average value is calculated for each safety integrity level to find the safety recommendations for design patterns.

The resulting average value may range between two integer numbers which makes the selection of the suitable recommendation more difficult. To solve this problem, we have introduced in [5] a new system of safety recommendations for design patterns. These recommendations include the original system of recommendations presented in IEC 61508 in addition to three new types: weakly not recommend, weakly recommended, and moderately recommended. The new recommendation types are classified based on the calculated average value as shown in Tab. 4.2.

Table 4.2.: Proposed system of recommendations for design patterns.

Recommendations		**Average Value (Avg)**
NR	Not Recommended	$Avg < 0.4$
WNR	Weakly Not Recommended	$0.4 \leq Avg \leq 0.6$
–	No Recommendation	$0.6 < Avg < 1.4$
WR	Weakly Recommended	$1.4 \leq Avg \leq 1.6$
R	Recommended	$1.6 < Avg < 2.4$
MR	Moderately Recommended	$2.4 \leq Avg \leq 2.6$
HR	Highly Recommended	$Avg > 2.6$

The method presented in this section, covers the first part of our approach for the assessment process of design patterns. It provides a systematic procedure to find safety recommendations that reflect the importance of the address pattern. Moreover, these recommendations and the intended safety integrity level give an indication of the severity of failures in the required application that will use this pattern.

By establishing the intended safety integrity level and finding the recommendations of the different applicable design patterns for this safety integrity level, the comparison process can be later conducted to find the most suitable pattern.

4.5.2. Relative Safety Improvement

Although safety and reliability of a safety-critical system are both highly related to the probability of failures, they differ on the type of failure, whether it is safe or unsafe. While reliability refers to the continuity of correct service, safety requirement addresses the severity of failures. Therefore, most of the safety design methods usually concentrate on dealing with unsafe failures[8].

Since the aim of this section is to provide a parameter that can be used in our approach to cover the second part, frequency of hazardous event, of the general risk metric shown in Equation (4.2), we will consider the probability of unsafe failure (P_{UF}) in our metric for risk assessment.

As shown in the reliability assessment, it is difficult to find actual values for this probability due to the abstract nature of design patterns. Hence, this probability will be calculated, like in reliability assessment, relative to the probability of unsafe failure in the basic system. If the basic system has a given reliability (R), then we will consider the probability of failure to be ($P_F = 1 - R$). These failures can be divided into two groups: safe and unsafe failures. For any system, the worst case occurs when all failures are unsafe; thus we assume all failures in the basic system to be unsafe to indicate the worst case.

To cover the second factor of the risk metric, we have proposed in [4] a new metric called *Relative Safety Improvement (**RSI**)*. This metric is defined as “the percentage improvement in safety (reduction in probability of unsafe failure) relative to the maximum possible improvement which can be achieved when the probability of unsafe failure is reduced to the minimum possible value (0)”.

Based on this definition, the relative safety improvement for a design pattern can be expressed as shown in Equation (4.3).

$$\begin{aligned} RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\ RSI &= (1 - \frac{P_{UF(new)}}{P_{UF(old)}}) \times 100\% \end{aligned} \tag{4.3}$$

– RSI: *Relative Safety Improvement.*
– $P_{UF(old)}$: *Probability of unsafe failure in the basic system.*
– $P_{UF(new)}$: *Probability of unsafe failure in the design pattern.*

In general, the combination of this metric and the previous system of safety recommendations provides a systematic method to assess design patterns for safety-critical systems. This method can be summarized as follows:

[8]**Unsafe (dangerous) failure**: a failure which has the potential to put the system in a hazardous state [46].

- The first step is to find the intended safety integrity level for the required application, or to conduct the next two steps for all safety integrity levels to provide a general assessment.
- The next step is to determine the safety recommendations which show the applicability of the design pattern under consideration to the safety integrity level(s).
- The final step is to compute the relative safety improvement which gives an indication about the safety improvement that can be achieved by this pattern. Furthermore, the safety metric can be used to investigate the effects of the individual parameters that affect the success of design patterns.

Finally, the relative safety improvement metric and the relative reliability improvement can be used in an analogous way. They can be computed using a simulation method (see Chapter 5) or through the employment of a mathematical model for design patterns as being done in our catalog of design patterns (see Chapter 7, 8, and 9).

5. A Case Study: Software Diversity

This chapter describes a case study conducted to illustrate safety and reliability assessment methods for design patterns. A Monte Carlo based simulation method has been used in this case study to demonstrate the use of the safety and reliability metrics by making an assessment for four software design patterns that share the concept of diversity programming.

This chapter includes two parts: the first part presents the basic concept of diversity programming, effects of a good acceptance test, and the basic idea of the recovery block with backup voting pattern (Section 5.1-5.4), while the second part includes the simulation procedure, reliability and safety assessment of software patterns (Section 5.5-5.7).

The basic idea of the recovery block with backup voting method was published in [7, 8], and part of the simulation results of reliability and safety assessment were published in [4, 7].

5.1. Concept of Diversity Programming

Software faults that may remain after software development process represent one of the critical reasons for unsafe failures in safety-critical system. Therefore, *Diversity Programming* or *Software Design Diversity* has been used for a long time as a major fault tolerance method in the design of software for safety-critical systems. It involves the independent generation of $N \geq 2$ functionally equivalent software modules called *versions* from the same initial specification by N independent teams of programmers. These versions can be executed in different ways depending on the used architecture. Moreover, the concept of diversity programming can be integrated with other techniques, like fault masking and fault detection, to construct different design methods.

Consequently, many design methods, that use the principle of software design diversity, have been proposed to be used as fault tolerance methods in the design of safety-critical systems. In our catalog of design patterns, we have included five design methods, which share the concept of software diversity, as design patterns (see Chapter 8).

In this case study, the following design methods were simulated and assessed to illustrate the basic idea of the proposed safety and reliability assessment metrics:

- ***N-Version Programming (NVP)*** [13]: *NVP* is the most well known fault-tolerant software method based on software diversity and fault masking. It runs the diverse versions in parallel on *N* hardware modules and uses a voting technique to perform a fault masking.
- ***Recovery Block (RB)*** [84]: in *RB*, which is based on fault detection, the independent software versions are executed sequentially followed by an acceptance test (*AT*) on a single hardware component. After the execution of the first version, the acceptance test is executed to check if the outcome is reasonable and to detect any possible erroneous result. If the acceptance test is passed, then the outcome is considered as true. Otherwise, if a fault is detected, the system state should be restored to its original state and an alternate version will be invoked to repeat the same computations. This process is repeated until either one of the alternate versions passes the test and gives the result, or no more versions are available. In the last case, an overall system failure is reported to execute the available safety function.
- ***Acceptance Voting (AV)*** [10]: the *AV* pattern integrates the concept of voting from *NVP* with the acceptance test used in the *RB*. This pattern executes the diverse versions in parallel on *N* hardware modules. The output of each version is evaluated for correctness using an acceptance test, and only those results, that pass the acceptance test, are used by the voting algorithm to generate the final result. The goal of the *AV* is to mask the faulty outputs from participating in the voting process.
- ***Recovery Block with Backup Voting (RBBV)*** [7, 8]: *RBBV* is an extension to the classical *RB* pattern. It improves the reliability of the classical *RB* in those situations where it is difficult to construct an effective acceptance test. (see Section 5.2 and 5.4)

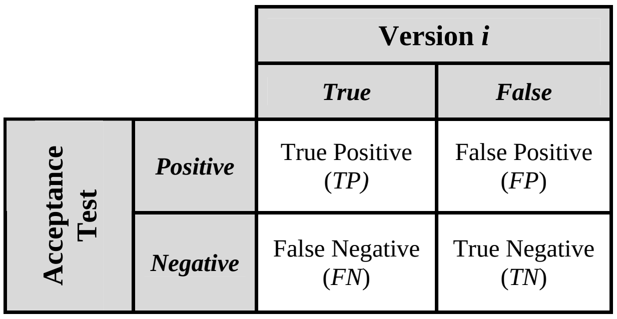
5.2. Acceptance Test

The acceptance test is a fault detection mechanism that can be used in fault tolerance design. In this fault detection method, the program is executed and the result is subjected to a test to confirm that the result is reasonable and to verify the correctness of calculations based on the software specifications. If the result passes the test, it is considered to be true. Otherwise, it is an indication of a fault.

The acceptance test method can be applied to a single module or to the complete software. It may have several components and may include checks for runtime errors [97] and mechanisms for implicit error detection.

In general, the relationship between the outcome of the tested version and the result of the acceptance test is shown in Fig. 5.1. It can be summarized in four possible cases:

1. ***True Positive (TP)***: The outcome of a version is correct and the acceptance test reports that the outcome is valid.
2. ***True Negative (TN)***: The outcome of a version is erroneous and the acceptance test reports that the outcome is invalid.
3. ***False Positive (FP)***: The outcome of a version is erroneous and the acceptance test reports that the outcome is valid.
4. ***False Negative (FN)***: The outcome of a version is correct and the acceptance test reports that the outcome is invalid.



		Version *i*	
		True	***False***
Acceptance Test	***Positive***	True Positive (*TP*)	False Positive (*FP*)
	Negative	False Negative (*FN*)	True Negative (*TN*)

Figure 5.1.: Relationship between the outcome of a version and acceptance test.

5.3. The Quality of the Acceptance Test

The quality of an acceptance test is denoted by the ability to identify the correct and erroneous outputs. Therefore, the probability of *true positive* and probability of *true negative* cases are highly desirable to be close to the ideal case where the output can be successfully identified. Conversely, the weaknesses in an acceptance test can be classified into two types: the *false positive* cases which include failure in error detection and the *false negative* cases which include incorrect identification of some conditions as errors.

The effectiveness of an acceptance test is considered as a major factor to the success of the fault tolerance methods that use the concept of acceptance test as in the recovery block and acceptance voting method. Consequently, it was shown in many studies [50, 85, 95] that the acceptance test should be carefully designed, reasonably simple, highly reliable, and with a high error detection coverage.

The competing goals of the desired acceptance test make the construction of an effective acceptance for some safety-critical systems very difficult. This difficulty will be reflected in a weakness in the acceptance test, which may then become a source of failure and may affect negatively the system safety and reliability. The effects of the two types of weakness in an acceptance test vary between safety and reliability. While false negative cases only affect reliability, the false positive cases affect both safety and reliability.

In order to alleviate the effect of a weak acceptance test, the concept of diversity can be applied to the acceptance test. In this case, multiple acceptance tests are independently developed by different teems to be used in conjunction with diverse software versions. This approach was applied in *N-self checking programming using acceptance test* [61, 59, 60], where the main difference of this method from the classical recovery block is the use of a separate acceptance test for each version (block).

In multiple acceptance test approach, the probability of concurrent failures in the acceptance test will be less than in the case of a single acceptance test, which will improve safety and reliability. On the other hand, constructing N diverse effective acceptance tests will be more difficult and costly than in the case of a single acceptance test.

5.4. Recovery Block with Backup Voting

The effect of false negative cases can be shown clearly in the classical recovery block method with a single acceptance test. In the recovery block method, if all the versions fail to pass the acceptance test, an overall system failure will be reported. The reason for this failure can be one of the following two cases:

1. All the versions fail concurrently to produce a correct result (*correlated failures*) and the acceptance test detects these failures.

2. There are some correct outcomes from some versions, but there is a failure in the acceptance test, which represents a false negative case in the acceptance test.

In the first case, the correlated failures are not related to the used acceptance test. It can be only improved by employing more techniques to increase the level diversity in software versions. To solve the second problem, we have proposed in [7] a new hybrid method called *Recovery Block with Backup Voting (RBBV)*.

The new method is shown in Fig. 5.2, and it was introduced to improve software reliability in the case of a weak acceptance test. It is based on the idea that when an overall failure is reported in the classical recovery block method, it is more likely that the acceptance test is the reason of this failure.

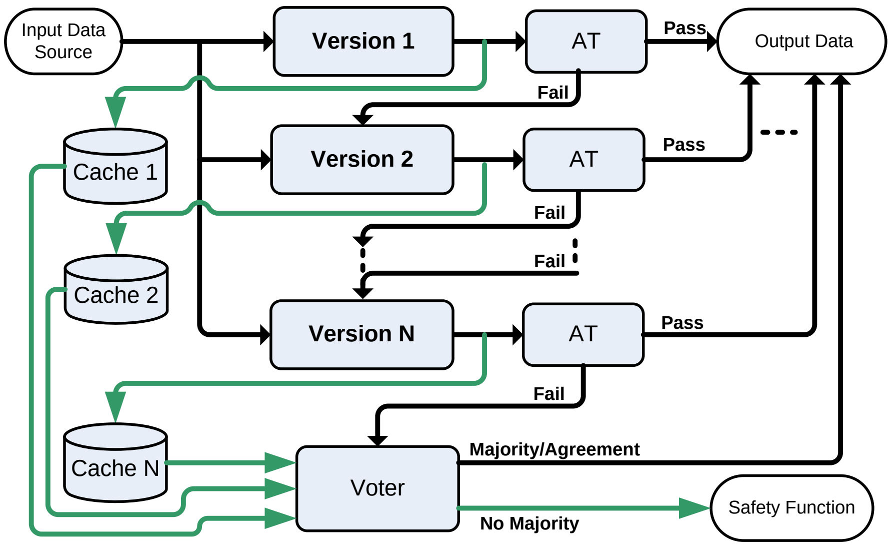


Figure 5.2.: Recovery block with backup voting method.

In this method, when a version fails to pass the acceptance test, the outcome is stored as a backup in a cache memory location to be used later. When the last version fails to pass the acceptance test, the system involves the following two conditions:

1. All the versions were executed, and the different outcomes are ready for use without using parallel hardware components as in the basic *NVP*.

2. All the versions fail to pass the acceptance test, and a voting technique might be more suitable than an acceptance test.

In this case, the stored backup results will be used as inputs to a voting technique as a last attempt to deliver a valid result. If the voter finds a majority between the backup values, it will consider this value as a valid result. Otherwise, if there is no majority, the voter will consider that the probability of the true negative is greater than the false negative case, which gives an indication that the problem includes correlated failures and not in the acceptance test. Consequently, a signal is generated to activate the available safety function.

5.5. Simulation

The section describes the simulator module, which we have developed to demonstrate the use of the safety and reliability metrics in this case study. An iterative simulation method, based on the *Monte Carlo* simulation method, was used to construct our simulator for reliability and safety assessment. The *Monte Carlo* simulation method, which relies on repeated generation of random numbers, has been widely used for reliability estimation in complex systems (see e.g. [70]).

5.5.1. Architecture

The architecture of the developed simulator is shown in Fig. 5.3. It consists of the following two modules:

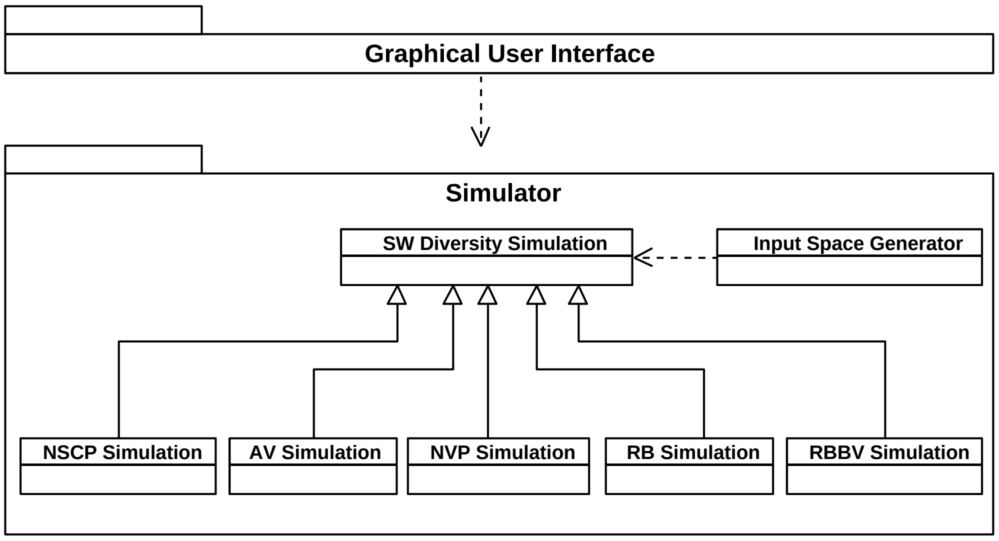


Figure 5.3.: The architecture of safety and reliability simulator.

- **Graphical User Interface(*GUI*)**: The *GUI* handles the interaction with the user, by getting the input parameters for simulation and displaying the simulations results to the user. The input parameters includes: the assessment method (*safety or reliability*), the number of the acceptance test (*single or multiple AT*), the type of simulation, and the probability settings for the diverse versions and the acceptance test.
- **Simulator**: The simulator module is the main module and it includes the following classes:

– A general base class that provides the basic variables and simulation methods for any design method that uses the concept of SW diversity.

– Five derived classes that inherit the base class and provide instant implementations for five different fault tolerance design methods. These classes cover *N-version programming, acceptance voting, N-self checking programming , recovery block, and recovery block with backup voting* method.

– A class that serves to randomly generate the input spaces for the different versions. The generated input spaces depend on the number of versions and the input parameters collected from the *GUI*.

5.5.2. Simulation Procedure

The procedure for safety and reliability simulation includes input data collection, construction of input spaces, execution of the selected methods, and the displaying of the simulation result.

The first step in our simulation is to collect the input parameters from the *GUI* to determine the simulation methods to be used, the parameters to construct the input spaces, and the acceptance test setting. In the second step, an input space of size 10^6 is constructed by the input space generator for each version. Each element in an input space contains a Boolean variable to indicate if this version produces a correct or faulty result for this element. The fault-causing inputs are scattered randomly in an input space according to the given probability of failure. In order to provide independent executions, the last step is repeated in each iteration of simulation.

In the first part of this simulation, we assume that the alternate versions fail independently which is reflected by the independent generation of the fault-causing inputs for the alternate versions. Nevertheless, dependent (or correlated) faults can be modeled in our simulator by adding fault-causing inputs at identical positions in the input spaces of the alternate versions based on a given criteria. This criteria depends entirely on the number of correlated versions and the relative ratio of the correlated failures to the total number of failures.

The next step involves the execution of the selected simulation of the used design methods on the generated input spaces. The execution is repeated for the given number of iterations, where fresh input spaces are generated in each iteration. The average value for the results of these iterations is calculated to give the final simulation result.

The final step involves the display of the final result using the *GUI* module. It provides two display methods: a graph display for scanning simulation and a text display for single-value simulation.

5.5.3. Notations and Assumptions

In the next two sections, we will use the following notations and assumption:

Notations

f	probability of failure in a version.
N	number of alternate versions.
P_{TP}	probability of *true positive* cases in the acceptance test.
P_{TN}	probability of *true negative* cases in the acceptance test.
P_{FP}	probability of *true positive* cases in the acceptance test.
P_{FN}	probability of *false negative* cases in the acceptance test.
RRI	relative reliability improvement.
RSI	relative safety improvement.

Assumptions

1. All software versions are functionally equivalent.
2. All software versions have equal probability of failure.
3. A simple majority voting technique is used in the voter.
4. Multiple acceptance tests are considered to be independent. Nevertheless, the dependency between multiple acceptance tests could be modeled in our simulator in the same way as in the correlated alternate versions.

5.6. Reliability Assessment

This section illustrates how the proposed reliability metric can be used in the reliability assessment of design patterns. It covers the reliability assessment of the software design patterns under consideration and the effects of the main factors that influence the reliability of these patterns.

5.6.1. The effect of the "Probability of Failure" in a version

In order to evaluate the effect of failures in the alternate versions on reliability, the used methods were simulated for different values of probability of failure in a version (f), ranging from 0.0 to 0.1. The simulation was conducted for a single and multiple acceptance tests, and it covered different values for P_{TP} and P_{TN} which give an indication about the quality of the used acceptance test(s).

The main purpose of $RBBV$ was to improve the reliability of the classical RB method. To prove this idea, the reliability simulation results of the four methods are given in Fig. 5.4. As shown in the results, the new method ($RBBV$) gives higher reliability than the classical RB for single and multiple acceptance tests. The reliability difference between RB and $RBBV$ varies according to different factors that will be discussed in the next part, but in all cases, $RBBV$ involves a considerable improvement to the reliability of the classical RB method.

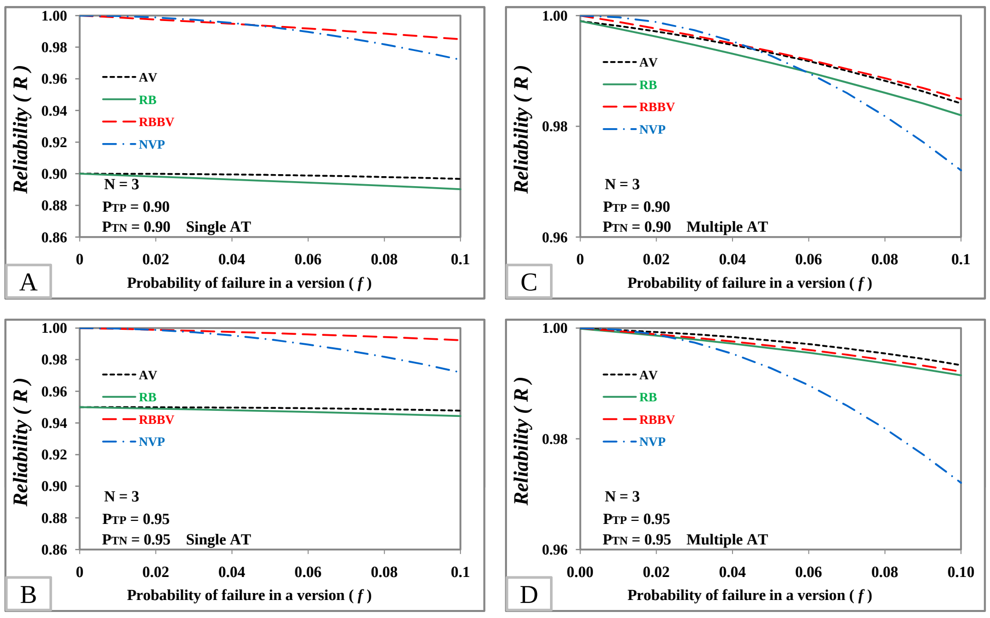


Figure 5.4.: Reliability improvement in $RBBV$.

To give a general reliability assessment using the proposed reliability metric, the simulation results for the relative reliability improvement of these patterns are shown in Fig. 5.5. For a single acceptance test with low probability of failure, $RBBV$ is resulting in higher RRI than RB and AV. Furthermore, its RRI is less dependent on the probability of failure in a version. At the same time, RB and AV do not give an improvement in RRI unless the probability of failure in the acceptance test is less than the probability of failure in the alternate versions as shown in Fig. 5.5-A and B. In these situations the single acceptance test becomes a source of single point of failure for RB and AV which decreases RRI.

For multiple acceptance tests or high value of P_{TP} and P_{TN}, which represent a good acceptance test, the difference between AV, RB and $RBBV$ becomes small as shown in Fig. 5.5-C,D,E and F. This aspect makes $RBBV$ more suitable for a situation that includes a single and weak acceptance test, while AV method is more suitable for multiple acceptance tests.

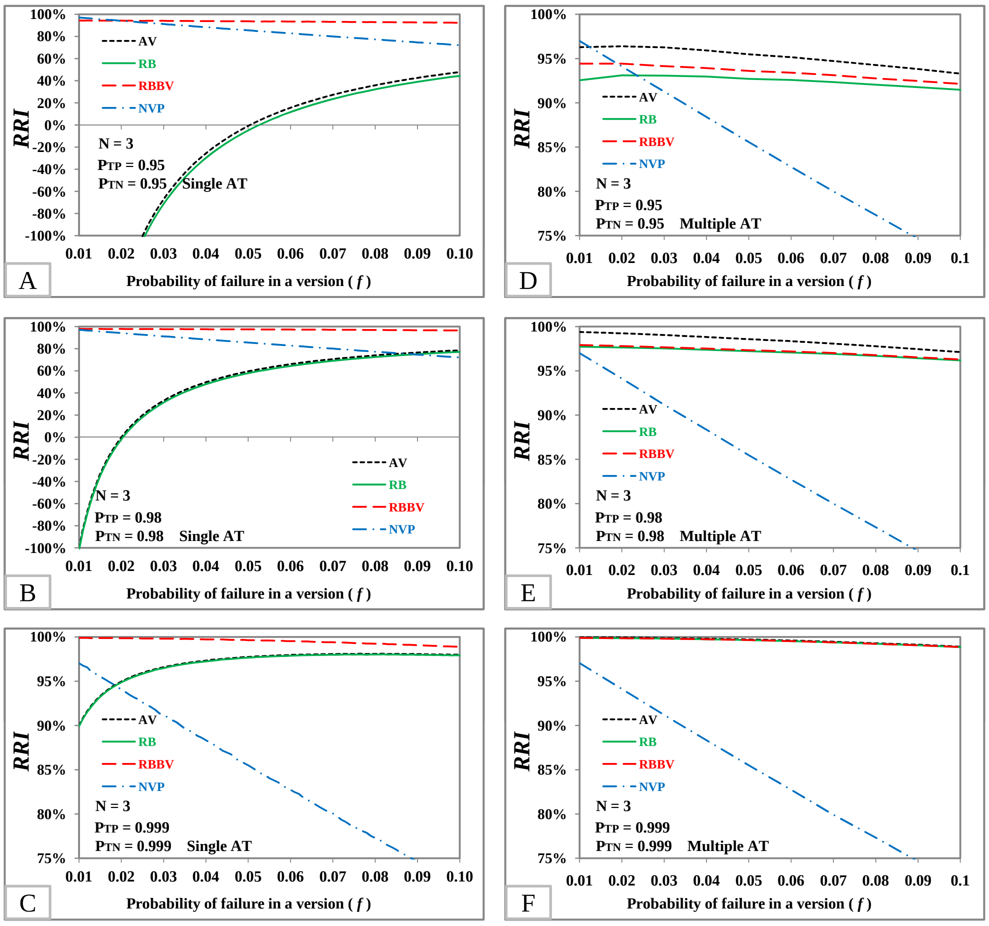


Figure 5.5.: The effect of the "Probability of Failure" in a single version on RRI.

5.6.2. The effect of the false negative cases in the acceptance test

As described above, the weakness in an acceptance test can be described by the failure to identify correct and erroneous results. In this part of simulation, the effect of false negative cases in the acceptance test is investigated in the methods that use acceptance tests, by varying the probability of false negative cases ($P_{FN} = 1 - P_{TP}$) from 0.0 to 0.1 for single and multiple acceptance tests.

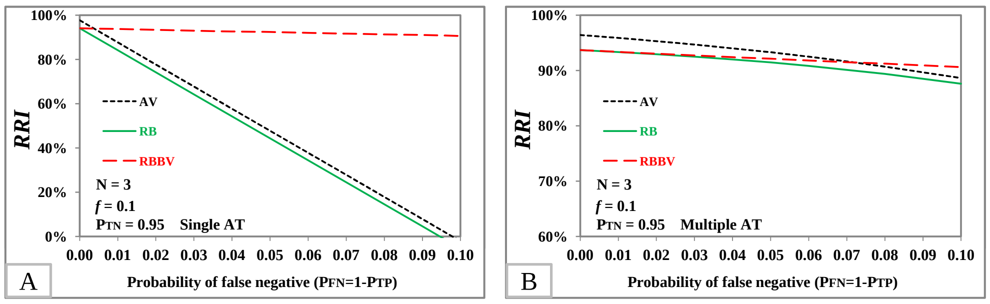


Figure 5.6.: The effect of the probability of false negative cases in AT on RRI.

The simulation result in Fig. 5.6-A shows that AV and RB involve high reliability degradation as P_{FN} increases, while the degradation in $RBBV$ is less dependent on P_{FN}. Fig. 5.6-B shows that for the case of multiple acceptance tests and a low value of P_{FN}, AV method gives higher RRI than RB and $RBBV$. Meanwhile, the high dependency of AV and RB on P_{FN} proves that $RBBV$ is more suitable for the situation that includes accepatnce test with high value of P_{FN}.

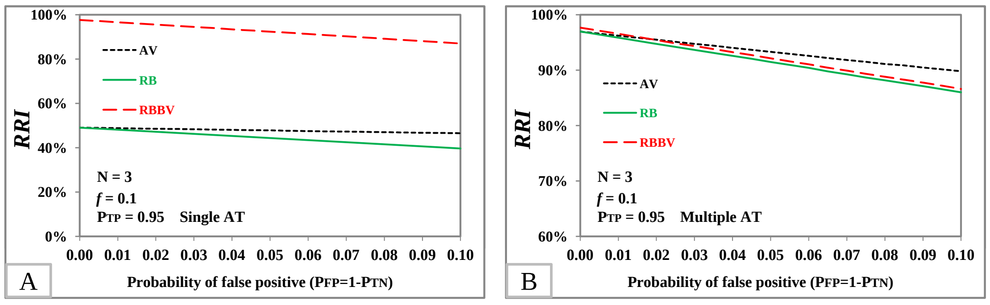


Figure 5.7.: The effect of the probability of false postive cases in AT on RRI.

5.6.3. The effect of the false positive cases in the acceptance test

To investigate the effect of the second type of weakness in the acceptance test, the simulation was conducted by varying the probability of false positive cases ($P_{FP} = 1 - P_{TN}$) from 0.0 to 0.1 for single and multiple acceptance tests.

The simulation result in Fig. 5.7 shows that AV, RB and RBBV suffer from this weakness in the same ratio, which makes the false positive cases with undiscovered faults a common problem to these methods.

5.7. Safety Assessment

This section illustrates the second proposed metric which is used to facilitate the safety assessment of design patterns. Like in reliability assessment, the effects of the probability of failures in aversion, the false negative cases, and false positive cases will be investigated.

5.7.1. The effect of the "Probability of Failure" in a version

The used patterns were simulated with different values of the probability of failures, ranging from 0.0 to 0.1, using a single and multiple acceptance tests. Furthermore, three different values for P_{TP} and P_{TN} were used in this part of simulation to give different levels of effectiveness in the used acceptance test(s).

Fig. 5.8 shows that NVP is resulting in small improvement in safety in comparison to the other methods, since it does not use fault detection by acceptance test. For simple voting strategy, it requires at least two versions to be correct in order to tolerate a single fault that may lead to unsafe failure. Conversely, AV is giving the highest safety improvement since it combines the advantage of fault detection by acceptance test and fault masking by voting. The acceptance test in AV masks the faulty results from participating in the voting process, which is resulting in a lower probability of unsafe failures. Likewise, the effect of the acceptance test on safety can be also seen in RB and $RBBV$ that give an intermediate value of *RSI* between NVP and AV.

In $RBBV$, the stored backup results may contain real faulty results due to concurrent failures in the alternate versions. While the probability of this case is very small, these faulty results may obtain a majority in the voting process, which leads to unsafe failure. This effect can be seen in the results as $RBBV$ is giving smaller *RSI* than in *RB*. Nevertheless, this small degradation in safety can be neglected in comparison to the great improvement in reliability that can be achieved by $RBBV$.

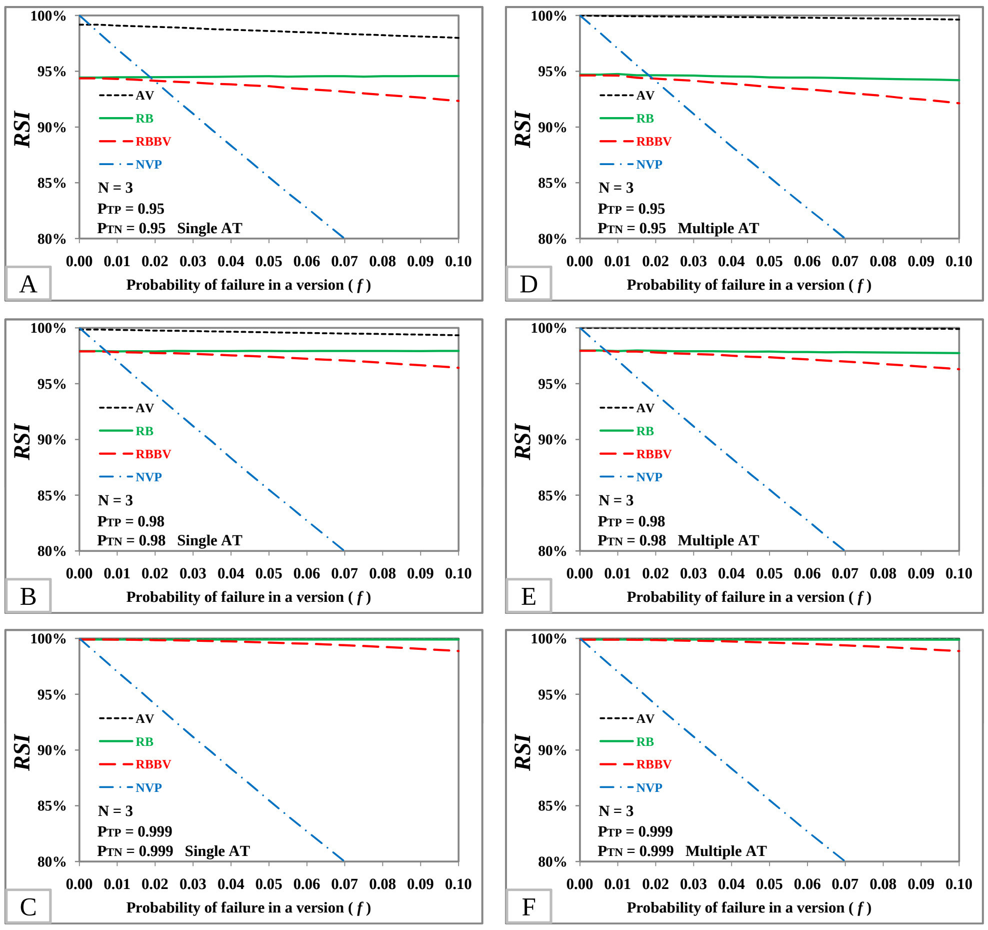


Figure 5.8.: The effect of the "Probability of Failure" in a single version on *RSI*.

5.7.2. The effect of the false negative cases in the acceptance test

Unlike the effect on reliability, the effect of the false negative cases on safety is very small as shown in Fig. 5.9. While the false negative cases have no effect on the safety of *RB*, they represent the main disadvantage to *RBBV* as discussed above. In *AV*, the false negative cases in the acceptance test prevent true outputs from participating in the voting process which may allow undetected faults to reach a majority with unsafe failure. However, the *RSI* degradation in *AV* as a result of false negative cases is less than in *RBBV*.

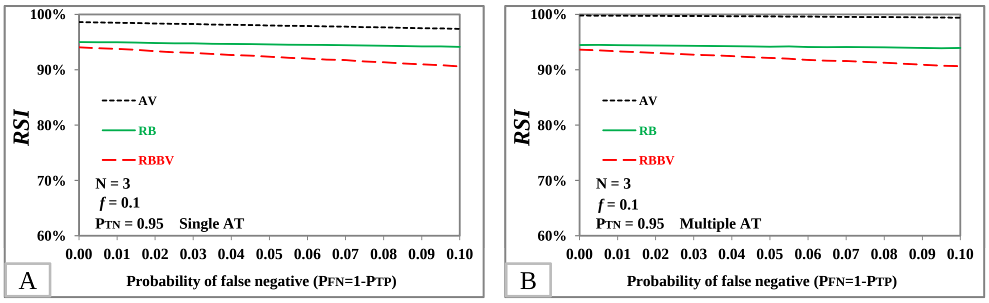


Figure 5.9.: The effect of the probability of false negative cases in *AT* on *RSI*.

5.7.3. The effect of the false positive cases in the acceptance test

The false positive cases are considered as a dominant factor to the safety improvement in the methods that are using acceptance tests for fault detection. As shown in Fig. 5.10, the false positive cases have identical effects on *RB* and *RBBV* since they include undiscovered faults leading to unsafe failures. On the other hand, *AV* has the least amount of dependency on the false positive case due to the integration of the fault detection and fault masking that has been discussed previously.

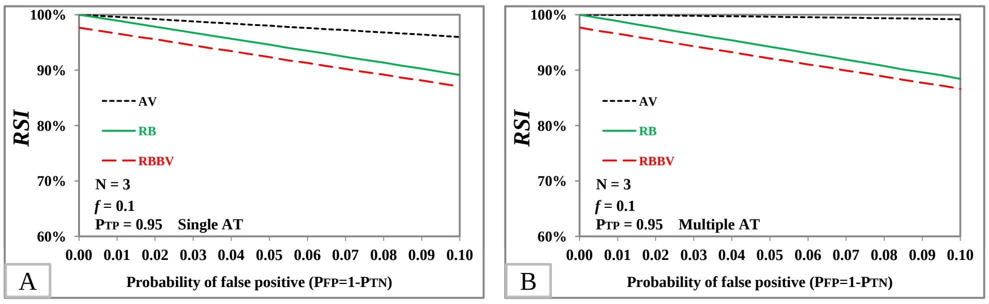


Figure 5.10.: The effect of the probability of false positive cases in *AT* on *RSI*.

5.8. Summary

In this case study, a simulation method was used to illustrate the proposed reliability and safety assessment method. The assessments were conducted for four design patterns that share the concept of diversity programming. The reliability and safety improvement that can be achieved by these patterns depend on many factors such as the use of single or multiple acceptance tests, and the quality of the used acceptance test.

The fault detection with an acceptance test presents a considerable safety and reliability improvement to the patterns that use this concept. In many applications, it is very difficult to construct an acceptance test, which makes NVP the best choice for these situations.

Conversely, when it is possible to construct an acceptance test, the selection of the suitable pattern depends on the quality of the acceptance test and the different importance between safety and reliability. For a single and weak acceptance test, $RBBV$ gives the highest reliability improvement, while AV gives the highest improvement for multiple acceptance tests. In terms of safety, AV gives the best safety improvement in all cases.

In general, the effects of the two types of weaknesses in an acceptance test vary between safety and reliability. While the false positive cases affect the reliability and safety of safety-critical software, the false negative cases affect the reliability more than safety.

In the applications that include a single hardware channel, $RBBV$ is resulting in higher reliability improvement than with RB in comparison to a small degradation in safety with $RBBV$. Nevertheless, the $RBBV$ method is still applicable to be a good choice for the situations that require a high reliability improvement and include weak acceptance tests with a considerable high probability of false negative cases.

6. Catalog of Design Patterns

Since the aim of this thesis is to construct a catalog of design patterns for safety-critical embedded systems, this chapter describes the catalog of design patterns tool which is the software version of our catalog. This catalog program is a .Net based tool that has been developed at the embedded software laboratory with the help of a diploma student using the C# programming language.

Section 6.1 introduces the main features of this catalog, while the description of the architecture is given in Section 6.2. Next, the data model for the database is given in Section 6.3. Section 6.4 and 6.5 present the main application guidelines and the decision support provided in this tool. Section 6.6 describes the implementation of the developed tool. Finally, the general components and the catalog structure are described in Section 6.7 and 6.8 respectively.

6.1. Features

In order to achieve the intended goals and requirements of this research, the catalog software provides the following features:

- An organized collection of design patterns stored in a database.
- Easy to use, install and remove.
- Providing two efficient pattern-retrieval methods:
  - PDF Viewer which provides a PDF copy of the intended pattern ready for printing.
  - An interactive browsing which makes it possible to navigate through contents of a pattern.
- Providing decision support to users for selecting a suitable pattern for the intended application by including an oriented search mechanism.
- Giving users the capability to create new design patterns.
- Giving users the capability to modify and update the current collection of patterns or the decision support components.
- Including an additional reliability and safety simulator for the patterns that are based on the principle of diverse programming.

6.2. Architecture

This section describes the architecture that has been used to develop the catalog program. To implement the required features for the current catalog, the software was divided into a set of modules where each module was implemented in a package. The package diagram which shows the architecture of this catalog and the dependencies between the packages is given in Fig. 6.1. The architecture includes the following modules:

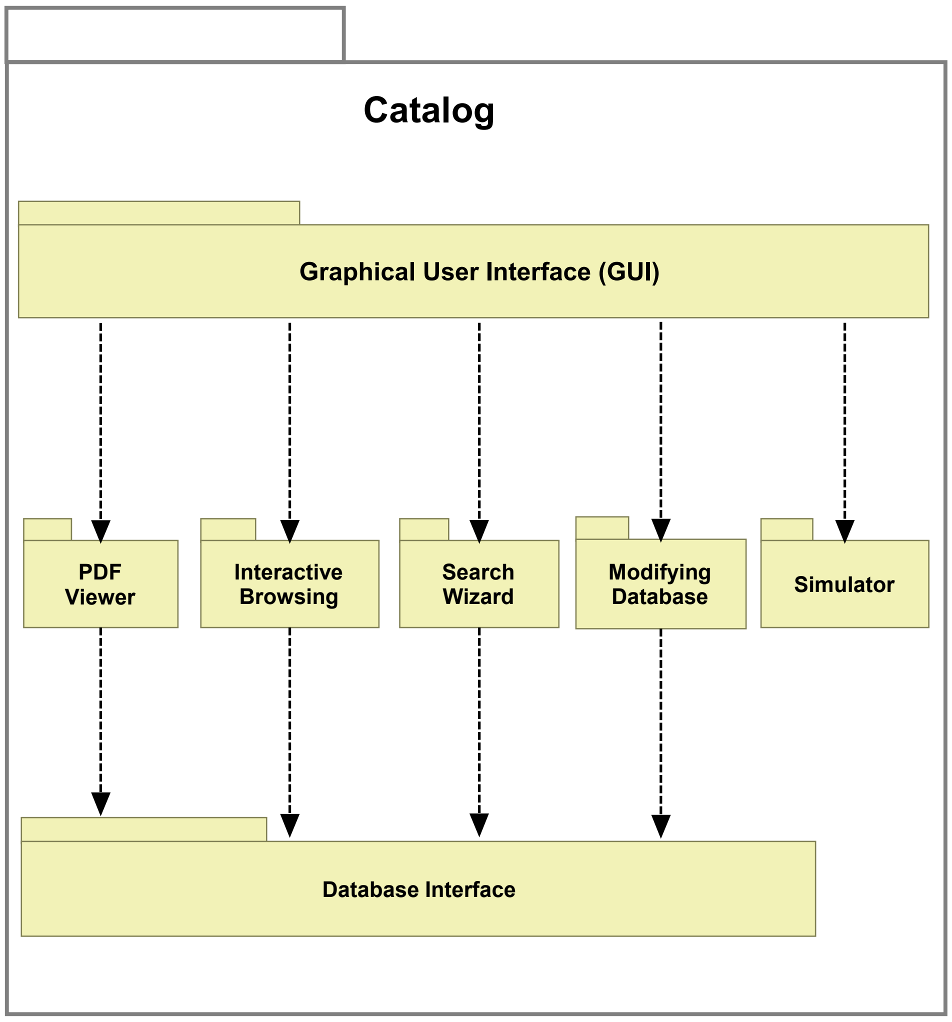


Figure 6.1.: The architecture of the catalog software.

Graphical User Interface (*GUI*)
The *GUI* represents an intermediate connection between the user and the other modules. It handles the interaction with the user by getting the user request and displaying the corresponding results. The *GUI* includes multiple interfaces customized for the different modules according to the required functionalities for these modules. For example, the *interactive browsing* interface provides a graphical display for the pattern structure and the ability to navigate the different components, while the *PDF Viewer* includes a plug-in to display the *PDF* files.

PDF Viewer
The *PDF Viewer* provides a list of *PDF* files for the available patterns, divided into three groups. Furthermore, it has been implemented using a *PDF plug-in*, which gives users the ability to browse, save, and print the patterns similar to the original *Adobe Acrobat Reader*[1] software.

Interactive Browsing
The *interactive browsing* module provides the second method for pattern presentation. It includes a selective navigation of the contents of the pattern by providing a detailed display for the pattern fields in addition to a graphical interactive display for the pattern structure. This feature gives users the ability to select, retrieve, and copy complete information about any part of the selected pattern.

Search Wizard
The *search wizard* module includes the decision support feature provided by this tool. It gives users the ability to find a suitable pattern or a combination of patterns for the desired application by answering questions in an oriented step by step wizard. Section 6.4 and 6.5 give more information about the decision support in this tool.

Modifying Database
This module serves two purposes. First, it allows users to modify the catalog contents, such as: the fields of the patterns, solutions, decision points (requirements), problems, and decision trees. Second, it provides the functionality to add new elements to the database such as creating a new pattern, a new solution, or a new decision tree. These two features offer an easy way to modify and extend the current catalog.

Simulator
The *simulator* module provides reliability and safety simulation for five

[1]http://www.adobe.com/products/reader/

software patterns that share the concept of software diversity. The architecture of this simulator was described previously in Section 5.5.1.

Database Interface
The module handles the interaction with the database by providing a set of functions to add, retrieve, update, and delete database records. It receives the requests from the other modules and performs the necessary database operations.

6.3. Data Model

This section presents the data model for our catalog which shows the data structure used for the catalog program and the relationship between the data elements. We use an *entity-relationship model (ERM)* [26] to describe our data model as shown in Fig. 6.2. The main entities that form our data model include the following:

- **Pattern**: The *Pattern* entity provides the basic information for the design patterns. It includes fields for all pattern elements, an image that represents the pattern structure, and a *PDF*-document for the pattern.
- **Part**: The *Part* entity includes a description of the elements that form the structure of the different patterns.
- **Solution**: The *Solution* entity provides the basic attributes of the available solutions that may contain a single or multiple patterns.
- **Requirement**: The *Requirement* entity provides the requirements that describe the application guidelines for the decision support and the possible values for these requirements.
- **Problem**: The *Problem* entity provides complete information about the problems that can be solved by the available patterns. Each entry contains a problem name, description, basic requirements, and a root node number for the corresponding tree.
- **Tree**: The *Tree* defines the decision tree for the different problems, where each problem has a single tree.
- **Node**: The *Node* entry defines the attributes of the nodes that construct the different decision trees.
- **Answers**: This entity defines the possible values for each requirement. These values will be used in the decision support to construct the answers at each decision point.

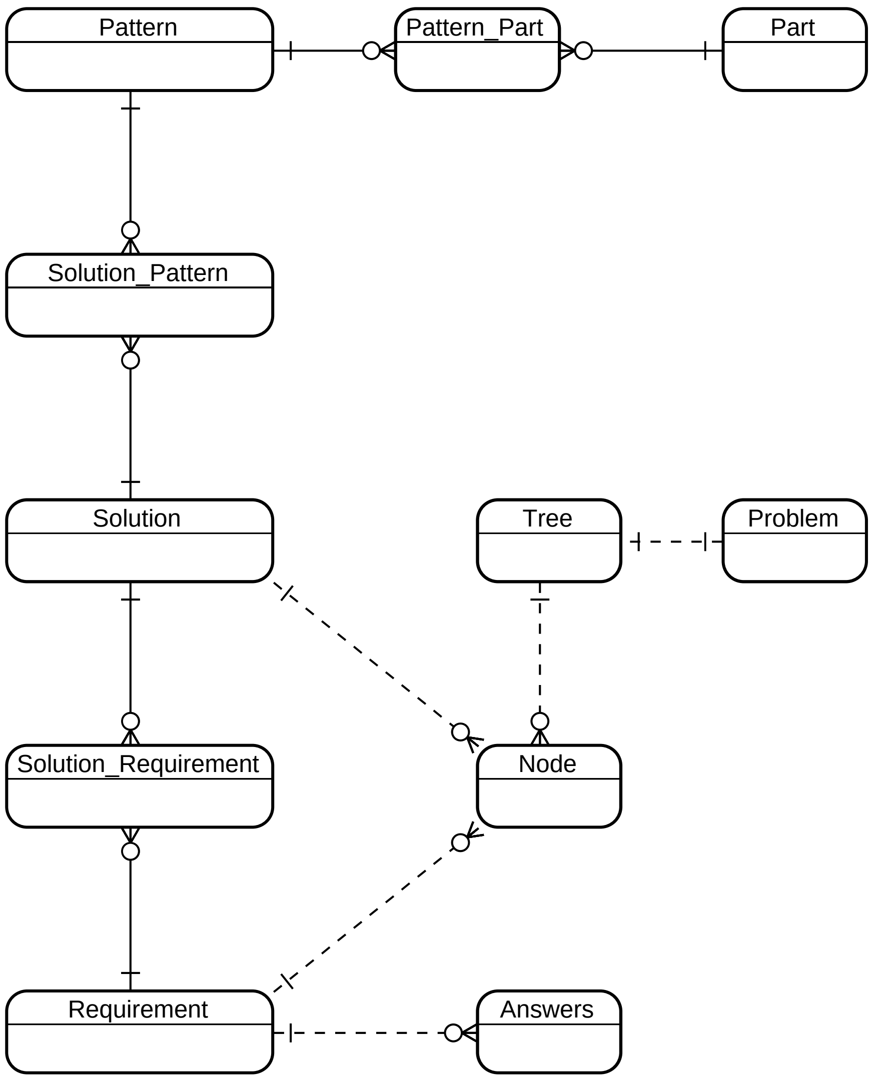


Figure 6.2.: The entity-relationship model for the database.

The main relationships that illustrate the associations between entities are:

- **Pattern_Part**: The *Pattern_Part* relationship relates the *Pattern* entity with the *Part* entity, specifying on which parts a pattern structure has been made.
- **Solution_Pattern**: The *Solution_Pattern* relationship relates the *Pattern* entity with the *Solution* entity, specifying the patterns that cooperate to form a solution.

- **Solution_Requirement**: The *Solution_Requirement* relationship relates the *Solution* entity with the *Requirement* entity to show the application requirements that should be satisfied for a solution.

In addition to the previous major entities and relationships, several minor entities and relationships have been defined to offer some minor functionalities such as the help files, interactive display, and patterns comparison.

6.4. Application Guidelines

The purpose of the decision support in our catalog, which will be discussed in the next section, is to present advices to users on how to select a suitable solution for a given design problem in the field of safety-critical embedded systems. This process requires the use of guidelines and decision points that characterize the situations in which the available patterns are appropriate.

In this context, the non-functional requirements for patterns, the limitations on the available resources, and the intended application requirements are used as decision points and guidelines for the selection process. For simplicity and to unify the notations for these guidelines, we called them requirements.

In the current version of the catalog software, we have fifteen requirements that are used in the search wizard. These requirements are:

1. **Hardware Redundancy**: Is it possible to provide multiple hardware channels?
2. **Software Diversity**: Is it possible to provide diverse software versions?
3. **Hardware Diversity**: Is it possible to provide heterogeneous hardware channels?
4. **Number of Hardware Channels**: The number of hardware channels which can be provided.
5. **Availability**: Investigates whether low or high level of availability is needed.
6. **Fail-Safe State**: Is there a reachable state that can be considered to be safe in the case of failure?
7. **Sequence Monitoring**: Is it necessary to provide a sequence monitoring?
8. **Acceptance Test**: Is it possible to construct an acceptance test to be used in conjunction with the software diversity?

9. **Quality of Acceptance Test**: Investigate whether the possible acceptance test can be effective or weak.

10. **Reliability Enhancement**: Is there a need for a reliability enhancement?

11. **Reliability vs. Safety**: Which non-functional requirement is more important: reliability or safety?

12. **Level of Safety Coverage**: Which level of safety coverage is needed: low, medium or high?

13. **Centralized Monitoring**: Is it necessary to provide a centralized monitoring and complex safety measures?

14. **Direct Safety Monitoring**: Is it necessary to provide direct safety monitoring?

15. **Fail-Safe Monitoring and Sequence Monitoring**: Is it necessary to provide a fail-safe state and sequence monitoring?

6.5. Decision Support

The users of the catalog of design patters might find the selection of a suitable pattern that solves a particular design problem, to be a hard process. Therefore, a systematic method, dedicated to help users, should be provided to ease the burden of decision making.

Our proposed solution for this difficulty is to develop a decision support system as a search wizard in our catalog. This approach involves three principles:

1. Providing the possible combinations of patterns as a set of solutions.

2. Grouping these solutions based on the problems that can be solved.

3. Providing a decision tree for each design problem.

6.5.1. Design Solutions

The pattern template, which has been presented in Section 3.4, provides a field for the related patterns that can be combined with the pattern under consideration to deal with related problems.

Instead of providing discrete patterns, the decision support in our catalog tool defines 33 solutions for the different problems, where each solution can represent:

- A single pattern solution like in *Triple Modular Redundancy* or *Sanity Check Pattern.*
- Multiple patterns solution, which contains a combination of closely related patterns like in *Recovery Block with Watchdog.*

6.5.2. Design Problems

In this tool, we have grouped the available solutions based on the design problems that can be solved by these solutions. The search wizard module provides solutions for the following problems:

- **A General Design Problem**: How to design a safety-critical embedded system.
- **Six Particular Problems**:
  - ***Transient Faults***: How to deal with transient faults, to provide some level of safety and reliability to the embedded system in an inexpensive manner.
  - ***Hardware Random Faults***: How to deal with hardware random faults and single-point of failure.
  - ***Hardware Systematic Faults***: How to deal with hardware systematic faults.
  - ***Safety Monitoring***: How to improve system safety through safety monitoring in the presence of a single point of failure.
  - ***Time Base Faults (Sequence Monitoring)***: How to identify time base faults by sequence monitoring.
  - ***Software Faults***: How to provide highly reliable fault-tolerant software for a highly safety-critical system.

6.5.3. Decision Trees

Each of the previously listed problems contains a set of applicable solutions, where each solution has its own requirements. In order to provide an oriented step-by-step approach that would lead the users through the process of solution selection, the solutions and their requirements are structured as decision trees, where each problem has its own decision tree. Fig. 6.3 shows the decision tree for a specific problem (Safety Monitoring), while a listing of all the decision trees can be found in Appendix A.

In each decision tree, the external nodes (*leaves*) contain the applicable solutions for the represented problem, and the internal nodes contain decision points that represent the requirements for these solutions. Moreover, every internal node has a number of children (*links*) corresponding to the number of possible values for the represented requirement, where a path from the starting node (*root*) to a leaf node (*solution*) gives the values of the requirements for this solution.

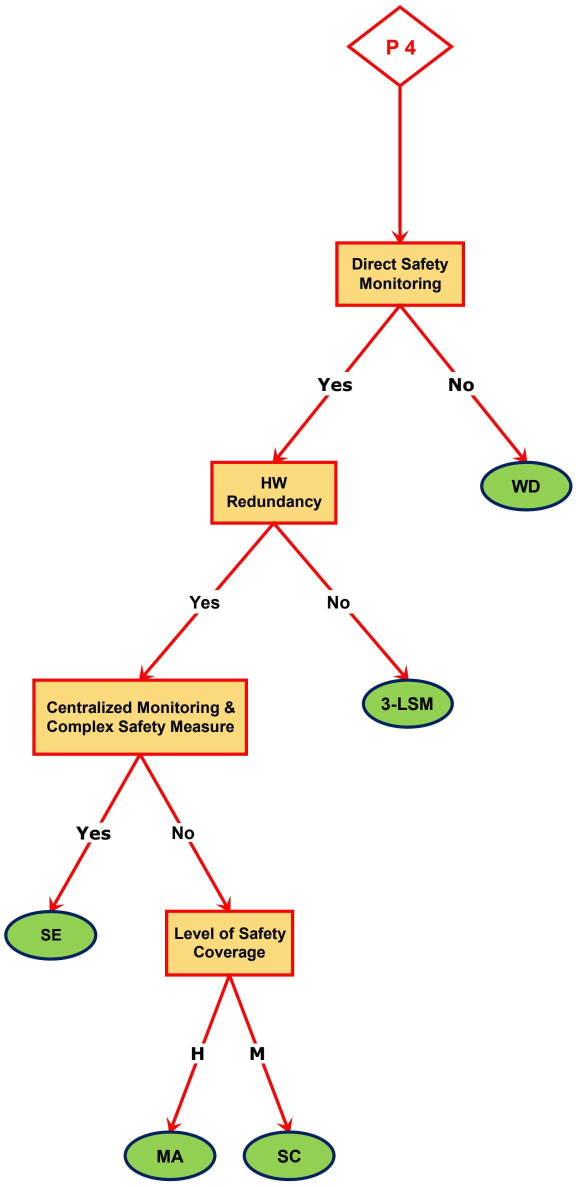


Figure 6.3.: The decision tree for the safety monitoring problem.

One of the main advantages of this approach is that the information contained in a decision tree is stored in a database, which makes the search program independent of the represented problems, requirements, and solutions. This feature eases the modification and extension of the decision support module to include: new patterns, solutions, problem, or decision trees without affecting the search program.

The search procedure for a given problem is conducted as an oriented sequence of questions to be answered by the user. It starts from the root node by constructing a question for the represented requirement, and then it provides the possible answers for this decision node. Normally, the user selects one of the possible answers, which will be checked by the procedure to determine the suitable child node for the next question. This process is repeated until the search procedure reaches a leaf node which is immediately selected to be an expected solution for the given problem. This answer represents the best available solution based on the requirements provided by the user, which implies that it is not the absolute best solution for this problem.

The search procedure gives the user the possibility to relax some requirements by providing a “don’t care” answer. In this situation, when the user relaxes a requirement at a decision node, the search procedure takes all the possible answers for this requirement into consideration by traversing all the out-going paths from this node.

The relaxation of some requirements during the search procedure might lead to multiple applicable solutions for the given problem. Therefore, the search procedure provides a comparison between the resulted solutions as a report. This report shows the applicable solutions as well as the requirements values of these solutions.

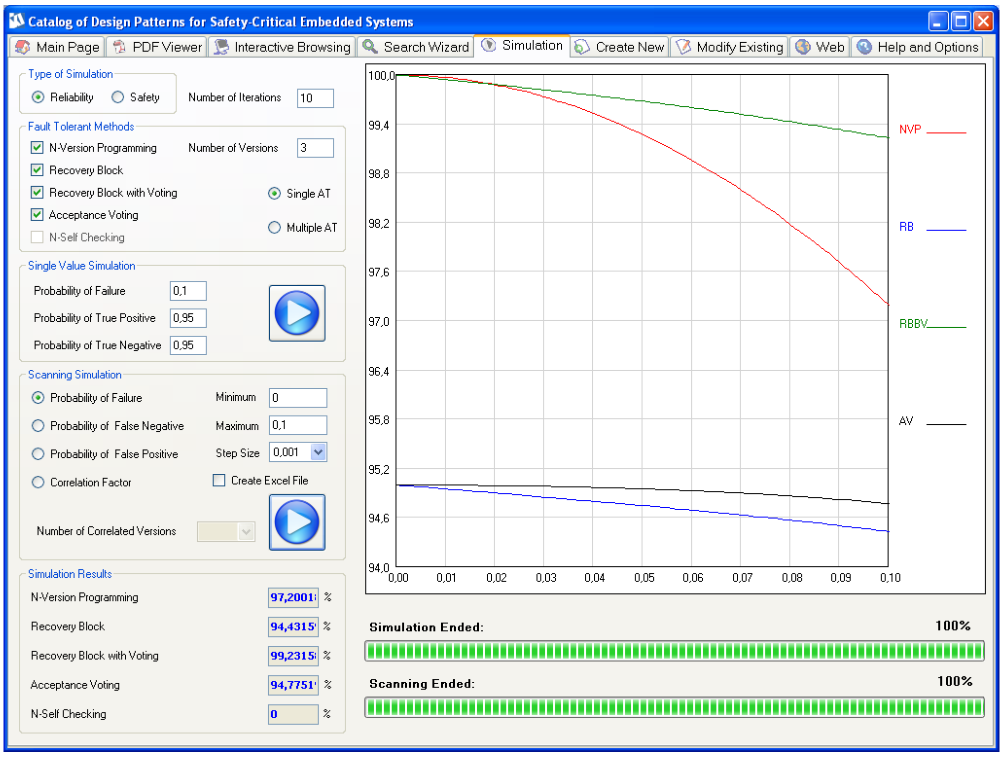
Note that the search procedure for a problem does not use all the existing requirements for two reasons: First, there are some requirements that are not related to the problem under consideration. Second, there are some basic default requirements that should be provided in all cases to solve a problem; like the software diversity for software faults handling, and the hardware redundancy for systematic faults handling. Therefore, the decision trees, which are provided in the decision support, include an organized minimal set of requirements to distinguish between the applicable solutions for each problem.

Summarizing, the developed search procedure aims to provide a simple decision support method that assists users in selecting a suitable pattern or a combination of related patterns to solve a particular problem with certain requirements. The representations of the existing patterns, requirements, problems, solutions, and relationships between these entities, as presented in our approach, facilitate this decision support. Furthermore, they increase the level of independence between the developed tool and the available patterns, which allows the expansion of the current collection of design patterns for safety-critical embedded systems.

6.6. Implementation

The catalog tool was implemented in two parts: the first part included the simulator which we have developed to conduct our case study, while the second part was implemented in a diploma thesis by *Grinin* [42] to embody the features described in Section 6.1.

The developed tool was implemented as a desktop application using Microsoft C# programming language, and SQL Server Compact 3.5 for the database. The graphical user interfaces of the simulator and the search wizard are shown in Fig. 6.4 and Fig. 6.5 respectively.



Type of Simulation
[✓ Reliability] Safety
Number of Iterations: 10

Fault Tolerant Methods
[✓] N-Version Programming
[✓] Recovery Block
[✓] Recovery Block with Voting
[✓] Acceptance Voting
[ ] N-Self Checking
Number of Versions: 3
[✓ Single AT] Multiple AT

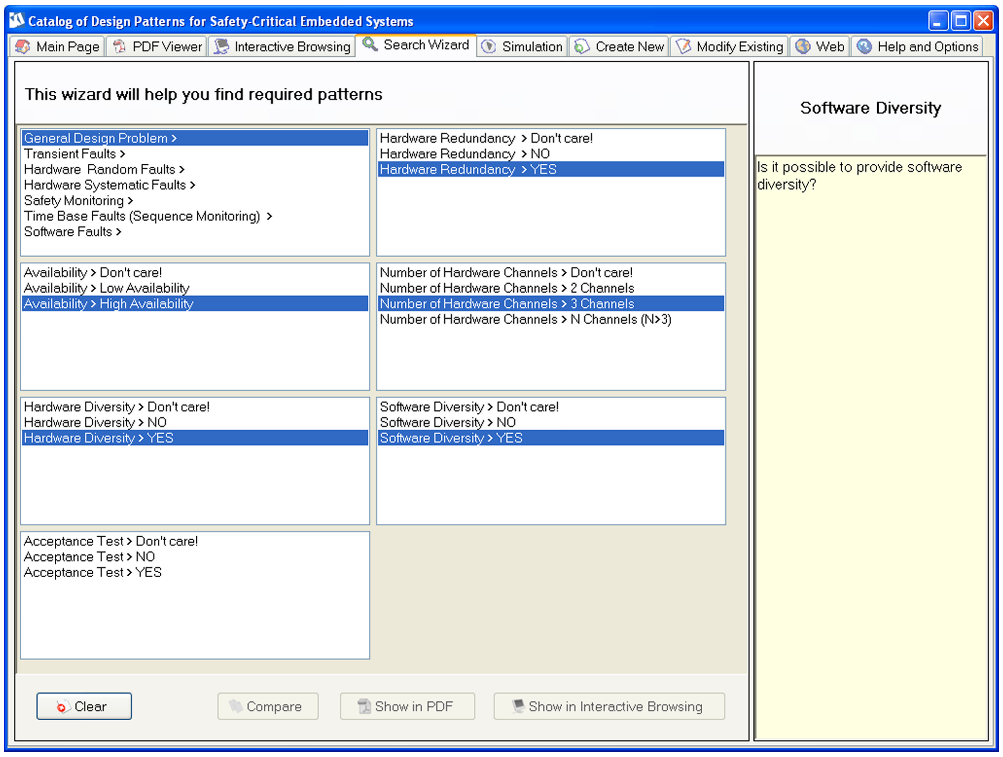
Single Value Simulation
Probability of Failure: 0,1
Probability of True Positive: 0,95
Probability of True Negative: 0,95

Scanning Simulation
[✓ Probability of Failure] Probability of False Negative / Probability of False Positive / Correlation Factor
Minimum: 0
Maximum: 0,1
Step Size: 0,001
[ ] Create Excel File
Number of Correlated Versions

Simulation Results
N-Version Programming: 97,2001 %
Recovery Block: 94,4315 %
Recovery Block with Voting: 99,2315 %
Acceptance Voting: 94,7751 %
N-Self Checking: 0 %

Simulation Ended: 100%
Scanning Ended: 100%

Figure 6.4.: Graphical user interface of the simulator module.


Catalog of Design Patterns for Safety-Critical Embedded Systems

This wizard will help you find required patterns

General Design Problem >
Transient Faults >
Hardware Random Faults >
Hardware Systematic Faults >
Safety Monitoring >
Time Base Faults (Sequence Monitoring) >
Software Faults >

Hardware Redundancy > Don't care!
Hardware Redundancy > NO
Hardware Redundancy > YES

Availability > Don't care!
Availability > Low Availability
Availability > High Availability

Number of Hardware Channels > Don't care!
Number of Hardware Channels > 2 Channels
Number of Hardware Channels > 3 Channels
Number of Hardware Channels > N Channels (N>3)

Hardware Diversity > Don't care!
Hardware Diversity > NO
Hardware Diversity > YES

Software Diversity > Don't care!
Software Diversity > NO
Software Diversity > YES

Acceptance Test > Don't care!
Acceptance Test > NO
Acceptance Test > YES

Software Diversity

Is it possible to provide software diversity?

Figure 6.5.: Graphical user interface of the search wizard module.

6.7. General Pattern

The next three chapters present the design patterns that form our catalog. In order to avoid multiple definitions for the common components in these patterns, we present in this section the basic system, which includes the general common components, as a general reference pattern.

The structure of the general pattern is shown in Fig 6.6 and it consists of the following general components:

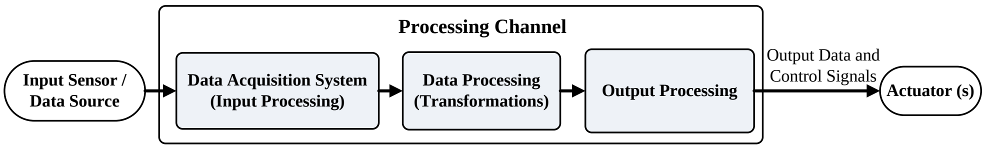

Processing Channel
Input Sensor / Data Source
Data Acquisition System (Input Processing)
Data Processing (Transformations)
Output Processing
Output Data and Control Signals
Actuator (s)

Figure 6.6.: The general pattern.

- *Processing Channel*: It represents a subsystem that performs some tasks in the complete system by taking an input data from an external source like a sensor, and performs some transformation on this data and then it uses the results to generate suitable command signals to activate some external actuators.
- *Input Sensor/Data Source*: It represents the source of information that is used as input to the designed system. Typically, this data comes either from the system user, or from external sensors used to monitor some actual environmental variables such as: temperature, pressure, speed, light, etc...
- *Data Acquisition (Input Processing)*: The input processing or the data acquisition unit collects the raw data from the input sensor and may convert the data to another form as in the analog to digital converter (*ADC*), and then it sends the data to the data transformation unit.
- *Data Processing (Transformation)*: This part may contain multiple data transformation components, where each one performs a single transformation or processing on the received data to execute the desired algorithm in order to generate the required control signals. The final component of this part sends the computed output to the output processing unit.
- *Output Processing*: The output processing unit takes the computed data from the data transformation unit and generates the final data and the control signals to activate the actuators. It can be considered as a device driver for the actuator.
- *Output Data and Control Signals*: The output data may contain some control signals to activate some actuators or messages for other components outside the system.
- *Actuator(s)*: The actuator represents the actual hardware that performs the action of the channel like: *motor, switch, heater*, or any other device that performs a specific function. There may be more than one actuator in a single channel.

6.8. Catalog Structure

The design patterns vary in their addressed problem, structure, and presented solution. Thus, we need a way to group them together in meaningful categories. In this catalog we classify the used design patterns into three groups, based on the type of pattern. The groups, which form our catalog, are:

- **Hardware Patterns**: Includes the patterns that contain explicit hardware redundancy to improve either reliability or safety. This group contains the following 8 patterns:
  - Homogeneous Duplex Pattern. (Section 7.3)
  - Heterogeneous Duplex Pattern. (Section 7.4)
  - Triple Modular Redundancy Pattern. (Section 7.5)
  - M-Out-Of-N Pattern. (Section 7.6)
  - Monitor-Actuator Pattern. (Section 7.7)
  - Sanity Check Pattern. (Section 7.8)
  - Watchdog Pattern. (Section 7.9)
  - Safety Executive Pattern. (Section 7.10)
- **Software Patterns**: Includes the patterns that use software diversity (software redundancy) to tolerate software faults. This group contains the following 5 patterns:
  - N-Version Programming Pattern. (Section 8.5)
  - Recovery Block Pattern. (Section 8.6)
  - Acceptance Voting Pattern. (Section 8.7)
  - N-Self Checking Programming Pattern. (Section 8.8)
  - Recovery Block with Backup Voting Pattern. (Section 8.9)
- **Combination of Hardware and Software Patterns**: Include 2 patterns that do not contain explicit hardware redundancy or software diversity:
  - Protected Single Channel Pattern. (Section 9.1)
  - 3-Level Safety Monitoring Pattern. (Section 9.2)

Clearly, there are many ways to categorize design patterns. For our purpose, it will be sufficient to use this structure, since it groups the patterns that address related problems and use related concepts.

7. Hardware Patterns

7.1. Notations

Table 7.1 shows the notations that will be used in this chapter for hardware patterns.

Table 7.1.: Notations for hardware patterns.

λ	Module failure rate
λ_r	Random failure rate
λ_s	Systematic failure rate
R	Reliability of each module (channel)
R_r	Reliability of the module against the random faults
R_s	Reliability of the module against the systematic faults
R_{comp}	Reliability of the comparator(fault detector)
C	Coverage factor which is defined as: the probability that the faulty module (channel) will be identified by the comparator [57]
R_{voter}	Reliability of the voter
R_{old}	Reliability of the basic system
R_{new}	Reliability after using the design pattern
RRI	Relative Reliability Improvement
P_{SF}	Probability of safe failure
$P_{UF(old)}$	Probability of unsafe failure in the basic system
$P_{UF(new)}$	Probability of unsafe failure after using the design pattern
RSI	Relative Safety Improvement

7.2. Redundancy Notes

While the general pattern gives the main common components, there is some redundancy of information in the presented patterns to provide complete and separate information on each pattern.

7.3. Homogeneous Duplex Pattern (HmD)

Other Names:
Homogeneous Redundancy Pattern [37], Standby-Spare Pattern, Dynamic Redundancy Pattern, Two-Channel Redundancy Pattern [44].

Type:
Hardware Pattern.

Abstract:
It is a hardware pattern that is used to increase the safety and reliability of the system by providing a replication of the same module (Modular redundancy) to deal with the random faults. The Homogeneous Duplex Pattern consists of two identical modules (channels); The primary (active) module and secondary (standby) module. This pattern can be applied to any level in the system design from a complete system (channel) to a single component. The idea of this pattern is based on the assumption that it is highly unlikely for two identical components to suffer a random fault simultaneously.

Context:
Developing an embedded system with no fail-safe-state in a situation that includes:
– High random failure rate and low systematic failure rate.
– There is a possibility to identify the faults that may lead to a system failure.
– High level of redundancy such as triple modular redundancy is not applicable.

Problem:
How to deal with random faults and how to make the system continue operating in the presence of a fault in one of the system components, in order to increase the safety and reliability of the system.

Pattern Structure:
The general structure for this pattern is shown in Fig. 7.1. This pattern can be applied to different levels: from a single component to a complete subsystem (channel). Generally, it consists of two identical modules; a primary (active) module and secondary (standby), and there is a fault detection unit that monitors the primary module and switches to the secondary module when a fault appears in the primary.
The function of each unit is as follows:

- *Module (Channel)*: (see the general pattern in Section 6.7).

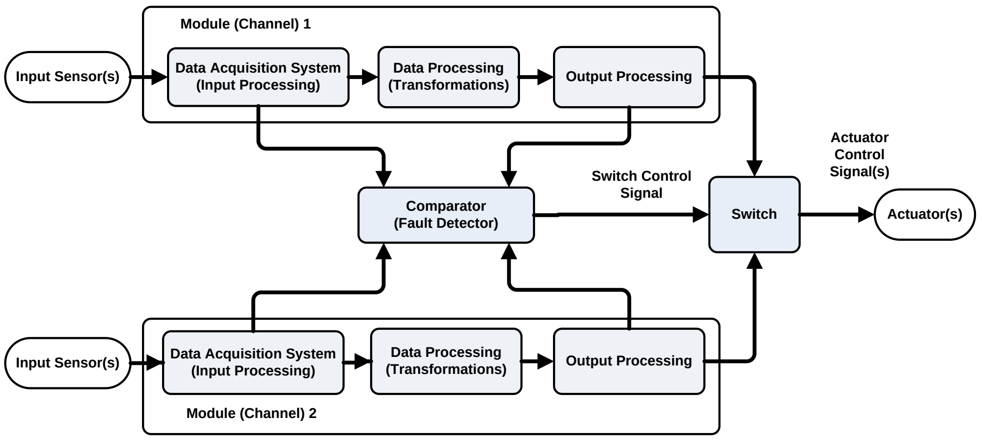

Module (Channel) 1
Input Sensor(s)
Data Acquisition System (Input Processing)
Data Processing (Transformations)
Output Processing
Actuator Control Signal(s)
Switch Control Signal
Comparator (Fault Detector)
Switch
Actuator(s)
Input Sensor(s)
Data Acquisition System (Input Processing)
Data Processing (Transformations)
Output Processing
Module (Channel) 2

Figure 7.1.: Homogeneous Duplex Pattern

- *Input Sensor(s)*: (see the general pattern in Section 6.7).
- *Actuator(s)*: (see the general pattern in Section 6.7).
- *Data Acquisition (Input Processing)*: (see the general pattern in Section 6.7).
- *Data Processing (Transformation)*: (see the general pattern in Section 6.7).
- *Output Processing*: (see the general pattern in Section 6.7).
- *Comparator (Fault Detector)*: The main idea behind this component is to identify the faulty module (channel) in the presence of a fault in one of the two channels, and to generate the instruction that controls the switch circuit. During the normal operation, the two channels give the same result, so the primary channel is used to do the required task, but when there is a fault in the primary channel, the comparator has to detect and to identify the faulty channel, then it generates an instruction to the switch circuit to switch from the primary channel to the secondary channel.

  The main problem in this component is how to identify the faulty module. Normally, this step requires information about the input data, the generated command to the actuators, and the state of the actuators from the two modules, in order to identify the module that suffers from a fault. There are many methods to identify the faulty channel such as:

  – *Acceptance Test* [57]: This method performs a check on the two channels by checking for input valid data within a given range and by checking the output signals from the two modules whether they are valid or not.

  - *Hardware Testing* [57]: It is possible to identify the faulty channel by subjecting both channels to some hardware/logic diagnostic test routines. This method is only efficient for permanent hardware faults, and it has low probability to identify the faulty channel in the presence of transient faults.
  - It is also possible to use a combination of the previous two methods; an acceptance test can be used as a first step to give an indication about the faulty channel, then a hardware testing routine can be used to confirm that the channel is faulty.

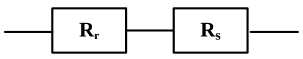
  There are two options for the comparator: either to use a single component that takes the information from the two channels which may become a source of single-point-failure, or to use two components, one for each channel.

- Switch: It is a simple component that takes the control information from the comparator to switch from the first to the second channel when there is a fault in the first channel, and vice versa.

Implication:
This section shows the implication of this pattern relative to the basic system.

- ***Reliability***:
  According to the nature of the fault, it can be classified into two types: *random* which is handled by this pattern and *systematic* which can not be handled here.

  For each module, the failure rate is equal to the sum of random and systematic failure rate:
  $\lambda = \lambda_r + \lambda_s$
  $\Rightarrow R(t) = e^{-(\lambda t)} = e^{-(\lambda_r+\lambda_s)t} = R_r R_s$
  Therefore, the reliability modeling will be represented as two serial components.



  Assume that the switch circuit is carefully designed with reliability ≈ 1. Then, the homogeneous duplex pattern will continue to work correctly as long as one of the two channels has no fault, but the two channels are identical and have the same architecture and the same systematic failure rate. So, the reliability modeling for this pattern will be as shown in Fig. 7.2.

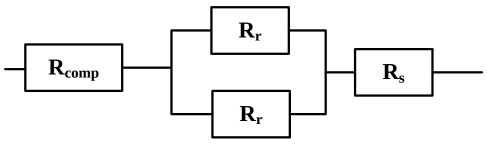


Figure 7.2.: Reliability modeling for HmD Pattern.

$$R_{new} = R_{comp}\left(R_r^2 + 2CR_r\left(1 - R_r\right)\right)R_s \tag{7.1}$$

Assume that the comparator was carefully designed with $R_{comp} \approx 1$.

$$R_{new} \approx \left[R_r^2 + 2CR_r\left(1 - R_r\right)\right]R_s \tag{7.2}$$

$\Rightarrow$ The percentage relative improvement in reliability is

$$\begin{aligned} RRI &= \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% \\ RRI &= \frac{\left(R_r^2 + 2CR_r\left(1 - R_r\right)\right)R_s - R_rR_s}{1 - R_rR_s} \times 100\% \end{aligned} \tag{7.3}$$

- ***Safety***:
  The homogeneous duplex pattern includes two design techniques: *test by redundant hardware* and *fault detection and diagnosis (comparator and acceptance test)*. According to the hardware and software requirements in the standard IEC 61508-2, 3 [46], the recommendations for these techniques are shown in Tab. 7.2.

Table 7.2.: Recommendations for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Test by redundant hardware	R	R	R	R
Fault detection and diagnosis (Comparator and Acceptance Test)	–	R	HR	HR

According to the last table, the average recommendations of this pattern for the different safety integrity levels are shown in Tab. 7.3.

Table 7.3.: Recommendations of HmD Pattern for safety integrity levels.

Pattern	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Homogeneous Duplex Pattern	WR	R	MR	MR

To compute RSI:

$$P_{UF(old)} = 1 - R = 1 - R_r R_s \tag{7.4}$$

The probability that the system will give an erroneous result that leads to unsafe failure is

$$P_{UF(new)} = 1 - R_{new} = 1 - R_{comp}\left(R_r^2 + 2CR_r\left(1 - R_r\right)\right)R_s \tag{7.5}$$

$\Rightarrow$ The percentage relative safety improvement is

$$\begin{aligned}
RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\
RSI &= \frac{\left(1 - R_{comp}\left(R_r^2 + 2CR_r\left(1 - R_r\right)\right)R_s\right) - \left(1 - R_r R_s\right)}{0 - \left(1 - R_r R_s\right)} \times 100\% \\
RSI &= \frac{R_{comp}\left(R_r^2 + 2CR_r\left(1 - R_r\right)\right)R_s - R_r R_s}{1 - R_r R_s} \times 100\% \qquad (7.6)
\end{aligned}$$

Under the assumption that the comparator was carefully designed with $R_{comp} \approx 1$.

$$RSI = \frac{\left(R_r^2 + 2CR_r\left(1 - R_r\right)\right)R_s - R_r R_s}{1 - R_r R_s} \times 100\% \tag{7.7}$$

It is clear from Equation (7.7) and Equation (7.3) that the relative safety improvement is equal to the relative reliability improvement.

- ***Cost***:
  The cost of this pattern can be classified into two parts:
  - *Recurring Cost*:
    - Since there are two identical channels, then the recurring cost will be increased by 200% comparing to the basic module.
    - The recurring cost will also include the cost of the comparator and fault detector component which depends on the type of implementation, in addition to the cost of the switch circuit.
  - *Development Cost*: The development cost for the two modules (channels) is equivalent to the development cost of a single channel, because they are identical.
- ***Modifiability***:
  In general, the level of modifiability of this pattern is similar to the basic system (single channel) since it consists of two identical channels. The only

difference exists in the comparator and fault detector. If it is required to modify the system, then the designer should take this modification into consideration and the comparator should be modified to use the new information in the fault detection process.

- ***Impact on Execution Time***:
  During the normal operation without faults, if the comparator circuit was implemented as a hardware circuit that is running in parallel with the two channels, then there is no change in the time of execution. But if this component includes software acceptance test, then the time of execution will be affected by the time needed to execute this test. In the presence of a fault, the execution time will depend on the speed of fault detection and the required time to replace the active channel with the standby channel.

Implementation:

- To implement this pattern, the computational channel should be duplicated as well as the other devices should be also duplicated.
- The comparator and fault detector component should be carefully designed to get a high coverage rate.
- Based on the state of the standby spare module, there are three modes:
  1. *Hot Redundancy*: The primary and the standby modules are working continuously in parallel to provide high reliability. This mode is suitable for application with strict time constraints.
  2. *Warm Redundancy*: The standby spare module runs in idle state and if the comparator and fault detector component detects any fault in the primary channel, the standby module takes over the load from the primary module.
  3. *Cold Redundancy*: In this arrangement, the standby spare module is normally off, and when a fault is detected, it will be put into operation immediately.
- The decision between the previous three arrangements depends on the required safety level, response time, and the power consumption. While hot redundancy has the highest safety level, cold redundancy has the least amount of power consumption. Thus, cold redundancy is more suitable for low power applications.

Consequences and Side Effects:

- The main drawback for this pattern is that it is not appropriate for systematic fault handling, since the two channels are identical and have the same possible fault. In the case of systematic fault, the switch circuit, that is designed to select between the two channels, will switch to one of the two faulty channels, which will lead to a complete system failure.
- When a fault is detected in the primary channel, the switch circuit switches over to the secondary channel which may include a loss of a computation step and may include a loss of input data, especially if there is no recovery time to redo the computation [37].

Related Patterns:

- As described previously, the main problem for this pattern is the systematic faults. So, it is possible to use heterogeneous duplex pattern which is similar to this pattern except in using heterogeneous modules (channels) instead of identical modules.
- To increase the safety and reliability of the system it is possible to use more than one spare channel which makes the pattern more expensive.

7.4. Heterogeneous Duplex Pattern (HtD)

Other Names:
Heterogeneous Redundancy Pattern [37], Diverse Redundancy Pattern.

Type:
Hardware Pattern.

Abstract:
It is a hardware pattern that is used to increase the safety and reliability of the system by providing a heterogeneous replication for the required module to deal with the random and systematic faults. The Heterogeneous Duplex Pattern consists of two independent different modules (channels); The primary (active) module and secondary (standby) module. This pattern is one of the most expensive patterns due to the high development cost. It can be applied to any level in the system design from a complete system (channel) to a single component. The two modules should be designed and implemented independently from each other.

Context:
Developing an embedded system with no fail-safe-state in a situation that includes:
– High random failure and high systematic failure rate.
– There is a possibility to identify the faults that may lead to a system failure.
– There is a possibility to use different designs and different implementation methods for the same module.

Problem:
How to deal with systematic faults as well as random faults, and how to make the system continue operating in the presence of a fault in one of the system components, in order to increase the safety and reliability of the system.

Pattern Structure:
In general, this pattern is similar to the homogeneous duplex pattern in the fact that it consists of two modules (channels) have the same functionality; a primary (active) module and secondary (standby), and there is a fault detection unit that monitors the primary module and switches to the secondary module when a fault appears in the primary. The main difference is that the two modules have independent designs or implementation methods, which gives this pattern the ability to handle systematic faults as well as random faults. The general structure for this pattern is shown in Fig. 7.3.

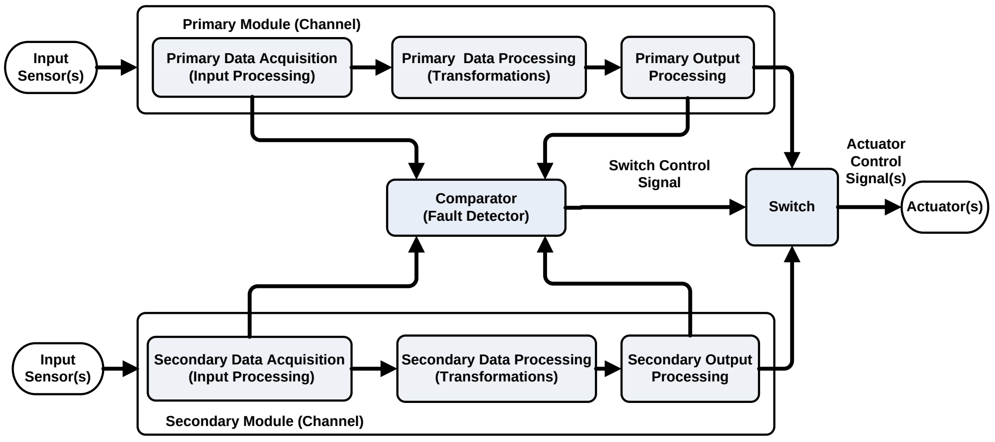

Primary Module (Channel)
Input Sensor(s)
Primary Data Acquisition (Input Processing)
Primary Data Processing (Transformations)
Primary Output Processing
Comparator (Fault Detector)
Switch Control Signal
Switch
Actuator Control Signal(s)
Actuator(s)
Input Sensor(s)
Secondary Data Acquisition (Input Processing)
Secondary Data Processing (Transformations)
Secondary Output Processing
Secondary Module (Channel)

Figure 7.3.: Heterogeneous Duplex Pattern

The function of each unit is as follows:

- *Module (Channel)*: Similar to the module described in the general pattern (Section 6.7). In this pattern, the two modules should be developed using different designs and different implementation techniques.
- *Input Sensor(s)*: (see the general pattern in Section 6.7).
- *Actuator(s)*: (see the general pattern in Section 6.7).
- *Data Acquisition (Input Processing)*: (see the general pattern in Section 6.7).
- *Data Processing (Transformation)*: (see the general pattern in Section 6.7).
- *Output Processing*: (see the general pattern in Section 6.7).
- *Comparator (Fault Detector)*: The main idea behind this component is to identify the faulty module (channel) in the presence of a fault in one of the two channels, and to generate the instruction that controls the switch circuit. During the normal operation, the two channels give the same result, so the primary channel is used to do the required task, but when there is a fault in the primary channel, the comparator has to detect and to identify the faulty channel, then it generates an instruction to the switch circuit to switch from the primary channel to the secondary channel.

  The main problem in this component is how to identify the faulty module? Normally, this step requires information about the input data, the generated command to the actuators, and the state of the actuators from the two modules, in order to identify the module that suffers from a fault. There are many methods to identify the faulty channel such as:

- *Acceptance Test* [57]: This method performs a check on the two channels by checking for input valid data within a given range and by checking the output signals from the two modules whether they are valid or not.
- *Hardware Testing* [57]: It is possible to identify the faulty channel by subjecting both channels to some hardware/logic diagnostic test routines. This method is only efficient for permanent hardware faults, and it has low probability to identify the faulty channel in the presence of transient faults.
- It is also possible to use a combination of the previous two methods; an acceptance test can be used as a first step to give an indication about the faulty channel, then a hardware testing routine can be used to confirm that the channel is faulty.

There are two options for the comparator: either to use a single component that takes the information from the two channels which may become a source of single-point-failure, or to use two components, one for each channel.

- Switch: It is a simple component that takes the control information from the comparator to switch from the first to the second channel when there is a fault in the first channel, and vice versa.

Implication:
This section shows the implication of this pattern relative to the basic system.

- ***Reliability***:
  According to the nature of the fault, it can be classified into two types: *random* and *systematic* which both can be handled by this pattern.

  For each module, the failure rate is equal to the sum of random and systematic failure rate:
  $\lambda = \lambda_r + \lambda_s$
  $\Rightarrow R(t) = e^{-(\lambda t)} = e^{-(\lambda_r + \lambda_s)t} = R_r R_s$
  Therefore, the reliability modeling for each channel will be represented as two serial components. The heterogeneous duplex pattern consists of two independent modules, and it will continue to work correctly as long as one of the two modules has no fault, neither random nor systematic.

Under the assumption that the switch circuit is carefully designed with reliability ≈ 1, the reliability modeling for this pattern will be as shown in Fig. 7.4.

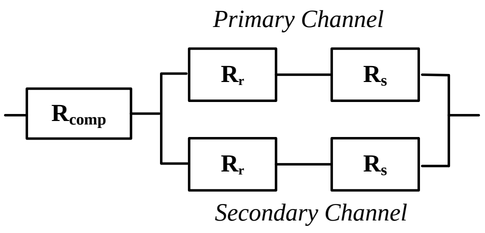


Figure 7.4.: Reliability modeling for HtD Pattern.

$$R_{primary} = R_{secondary} = R_r R_s = R$$

$$R_{new} = R_{comp}\left(R^2 + 2CR(1-R)\right) \tag{7.8}$$

Assume that the comparator was carefully designed with $R_{comp} \approx 1$.

$$R_{new} \approx R^2 + 2CR(1-R) \tag{7.9}$$

$\Rightarrow$ The percentage relative improvement in reliability is

$$\begin{aligned} RRI &= \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% \\ RRI &= \frac{R^2 + 2CR(1-R) - R}{1-R} \times 100\% \\ RRI &= R(2C-1) \times 100\% \qquad (7.10) \end{aligned}$$

- ***Safety***:
  The heterogeneous duplex pattern includes three design techniques: *diverse hardware*, *test by redundant hardware*, and *fault detection and diagnosis (comparator and acceptance test)*. According to the hardware and software requirements in the standard IEC 61508-2, 3 [46], the recommendations for these techniques are shown in Tab. 7.4.

  According to the last table, the average recommendations of this pattern for the different safety integrity levels are shown in Tab. 7.5.

  To compute RSI:

$$P_{UF(old)} = 1 - R \tag{7.11}$$

Table 7.4.: Recommendations for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Test by redundant hardware	R	R	R	R
Fault detection and diagnosis (Comparator and Acceptance Test)	–	R	HR	HR
Diverse hardware	–	–	R	R

Table 7.5.: Recommendations of HtD Pattern for safety integrity levels.

Pattern	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Heterogeneous Duplex Pattern	–	R	R	R

The probability that the system will give an erroneous result that leads to unsafe failure is

$$P_{UF(new)} = 1 - R_{new} = 1 - R_{comp}\left(R^2 + 2CR\,(1-R)\right) \tag{7.12}$$

$\Rightarrow$ The percentage relative safety improvement is

$$\begin{aligned} RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\ RSI &= \frac{\left(1 - R_{comp}\left(R^2 + 2CR\,(1-R)\right)\right) - (1-R)}{0 - (1-R)} \times 100\% \\ RSI &= \frac{R_{comp}\left(R^2 + 2CR\,(1-R)\right) - R}{1-R} \times 100\% \end{aligned} \tag{7.13}$$

Under the assumption that the comparator was carefully designed with $R_{comp} \approx 1$.

$$\begin{aligned} RSI &= \frac{\left(R^2 + 2CR\,(1-R)\right) - R}{1-R} \times 100\% \\ RSI &= R(2C-1) \times 100\% \end{aligned} \tag{7.14}$$

Similar to the homogeneous duplex pattern, the relative safety improvement is equal to the relative reliability improvement.

- ***Cost***:
  The cost of this pattern can be classified into two parts:
  - *Recurring Cost*:
    * Since there are two identical channels, then the recurring cost will be increased by 200% comparing to the basic module.

    * The recurring cost will also include the cost of the comparator and fault detector component which depends on the type of implementation, in addition to the cost of the switch circuit.

  – Development Cost: The high development cost of two diverse modules makes this pattern more expensive relative to other patterns. Sometimes, it is very expensive and difficult to provide two independent modules because this step will include two independent designs and two development teams to provide independence of systematic faults [37]. In general, the development cost for this pattern is equal to 200% comparing to the basic system.

- ***Modifiability***:
  As this pattern includes two independent and diverse modules, the level of modifiability is less than in the basic system (single channel). The two modules should be modified in order to add any new feature to the subsystem, which will duplicate the required efforts. There is another problem related to the comparator and fault detector; if it is required to modify the system, then the designer should take this modification into consideration and the comparator should be modified to use the new information in the fault detection process. These extra requirements will change the level of modifiability.

- ***Impact on Execution Time***:
  During the normal operation without faults, if the comparator circuit was implemented as a hardware circuit that is running in parallel with the two channels, then there is no change in the time of execution. But if this component includes software acceptance test, then the time of execution will be affected by the time needed to execute this test. In the presence of a fault, the execution time will depend on the speed of fault detection and the required time to replace the active channel with the standby channel.

Implementation:

- To implement this pattern, the computational channel should be duplicated as well as the other devices should be also duplicated.
- The duplicated modules should be implemented using independent designs or independent methods to avoid common systematic faults.
- It is common to use different sensors, actuators and computing hardware components with different hardware implementations and different technologies to duplicate the required module (channel) [37].

- It is more preferable to use different software versions that are designed by different teams and using different algorithms, when it is possible.
- The comparator and fault detector component should be carefully designed to get a high coverage rate.
- Similar to the Homogeneous Duplex Pattern, there are three modes for redundancy in this pattern: hot, warm, and cold redundancy.
- The decision between the previous three arrangements depends on the required safety level, response time, and the power consumption. While hot redundancy has the highest safety level, cold redundancy has the least amount of power consumption.

Consequences and Side Effects:

- The main drawback is the very high recurring and development cost for this pattern as shown in the previous part.
- When a fault is detected in the primary channel, the switch circuit switches over to the secondary channel which may include a loss of a computation step and may include a loss of input data, especially if there is no recovery time to redo the computation [37].

Related Patterns:
To increase the availability of the system, it is possible to use triple modular redundancy with heterogeneous modules as in this pattern, but this option includes a very high recurring and development cost, as three diverse modules should be provided instead of two.

7.5. Triple Modular Redundancy Pattern (TMR)

Other Names:
2-oo-3 Redundancy Pattern, Homogeneous Triplex Pattern.

Type:
Hardware Pattern.

Abstract:
This pattern is a variation of homogeneous hot redundancy, that consists of three identical modules operate in parallel to detect random faults, in order to enhance reliability and safety in a system with no fail-safe-state. The modules operate in parallel to produce three results that are compared using a voting system to produce a common result as long as two channels or more have the same result. This structure allows the system to operate and to provide functionality in the presence of a random fault without losing the input data [37].

Context:
Developing an embedded system with no fail-safe-state in a situation that includes high random failure rate and no limitation on redundancy, with the purpose of improving safety and reliability of the system.

Problem:
How to deal with random faults and single-point of failure in order to increase the safety and reliability of the system without losing the input data in the presence of faults.

Pattern Structure:
As shown in Fig. 7.5, this pattern contains three identical modules or channels operate in parallel. This structure is used to prevent the failure of a single component, which may lead to a complete system failure [97]. If a single fault occurs in one channel then the other two channels will continue to work correctly and produce the correct actuation control signals.

The voter plays a main role in this pattern by applying the voting policy to take the majority from the results which represents the correct actual result. This pattern does not identify the type or the reason of the fault; it just determines the module that contains a fault without correcting the fault itself.
The function of each unit is as follows:

- *Module (Channel)*: (see the general pattern in Section 6.7).
- *Input Sensor(s)*: It represents the source of information for this pattern

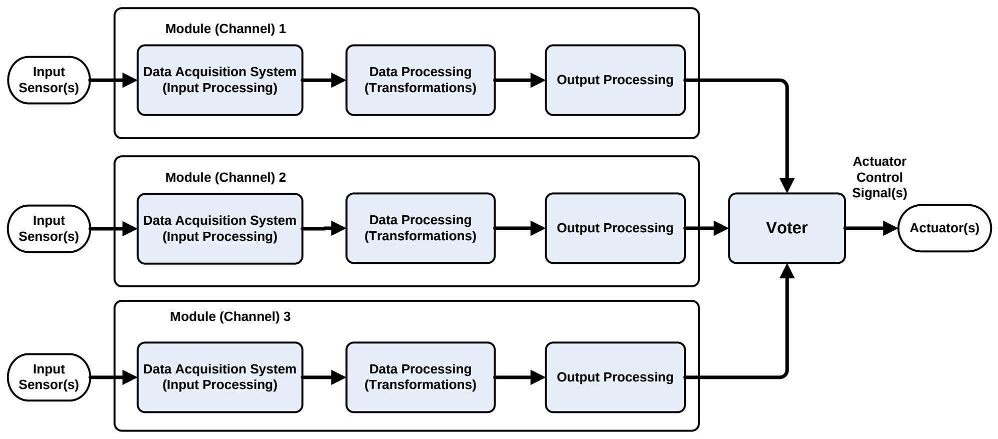


Figure 7.5.: Triple Modular Redundancy Pattern

(see the general pattern in Section 6.7). It is possible to use one sensor for the three modules, but in this case the failure of this sensor would affect inputs to all of the modules. Another choice is to use three separate sensors in order to remove the possible single-point of failure in this sensor.

- *Actuator(s)*: (see the general pattern in Section 6.7).
- *Data Acquisition (Input Processing)*: (see the general pattern in Section 6.7).
- *Data Processing (Transformation)*: (see the general pattern in Section 6.7).
- *Output Processing*: (see the general pattern in Section 6.7).
- *Voter*: The voter receives three outputs from the output processing units of the modules and implements a voting policy to find the majority output. If at least two modules give the same output, then the voter takes this result and discards the output from the deviated channel. Like the sensor, this component could be a reason for a single-point of failure. So, it may be possible to use three separate voters instead of a single one. Another possible solution is to take care of the design of this unit to produce a simple and reliable voter.

Implication:
This section shows the implication of this pattern relative to the basic system.

- ***Reliability***:
  The triple modular redundancy will continue to work correctly as long as

two or more channels have no fault.

$$R_{new} = R_{voter}\left(3R^2 - 2R^3\right) \tag{7.15}$$

Assume that the voter is a simple component that was carefully designed with reliability ($R_{voter} \approx 1$).

$$R_{new} \approx 3R^2 - 2R^3 \tag{7.16}$$

$\Rightarrow$ The percentage relative improvement in reliability is

$$\begin{aligned} RRI &= \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% \\ RRI &= \frac{3R^2 - 2R^3 - R}{1 - R} \times 100\% \\ RRI &= \left(2R^2 - R\right) \times 100\% \end{aligned} \tag{7.17}$$

- ***Safety***:
  The TMR pattern includes two design techniques: *test by redundant hardware*, and *fault detection and diagnosis (voter)*. According to the hardware and software requirements in the standard IEC 61508-2, 3 [46], the recommendations for these techniques are shown in Tab. 7.6.

Table 7.6.: Recommendations for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Test by redundant hardware	R	R	R	R
Fault detection and diagnosis (Voter)	–	R	HR	HR

According to the last table, the average recommendations of this pattern for the different safety integrity levels are shown in Tab. 7.7.

Table 7.7.: Recommendations of TMR Pattern for safety integrity levels.

Pattern	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Triple Modular Redundancy Pattern	WR	R	MR	MR

To compute RSI:

$$P_{UF(old)} = 1 - R \tag{7.18}$$

The triple modular redundancy will continue to work correctly as long as two or more channels have no fault. Thus, the probability that the system will give an erroneous result that leads to unsafe failure is

$$P_{UF(new)} = 1 - R_{new} = 1 - \left(3R^2 - 2R^3\right) \tag{7.19}$$

$\Rightarrow$ The percentage relative safety improvement is

$$\begin{aligned} RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\ RSI &= \left(1 - \frac{\left(1 - \left(3R^2 - 2R^3\right)\right)}{1 - R}\right) \times 100\% \\ RSI &= \left(2R^2 - R\right) \times 100\% \end{aligned} \tag{7.20}$$

Like in *HmD* and *HtD* pattern, it is clear from Equation (7.20) and Equation (7.17) that the relative safety improvement is equal to the relative reliability improvement.

- ***Cost***:
  The cost of this pattern can be classified into two parts:
  - *Recurring Cost*:
    - This pattern has a high recurring cost due to the using of three parallel modules. So, the recurring cost is 300% comparing to the basic system.
    - The cost of voter which is normally a simple hardware circuit that depends on the type of the output control signal and the implementation method.
  - *Development Cost*: The three modules (channels) are identical and using the same algorithm and the same software. Therefore, the development cost for this pattern is similar to the development cost of the basic system.
- ***Modifiability***:
  This pattern does not change the level of modifiability of the basic system, since if you want to modify the functionality for a system with *TMR*, the effort will be equivalent to modifying a simple channel.

  It is also possible to modify this pattern to *M-oo-N* redundancy by increasing the number of channel to increase the reliability of the system. This is an easy step that just includes the modifying of the voter to take the new channels into its voting policy.
- ***Impact on Execution Time***:
  This pattern has a little influence on the executing time comparing to the basic system, since the three modules are running separately in parallel. The only time influence of this pattern is the small delay of the voter hardware that affects the response time from reading the input signal to generating the control actuating signals.

Implementation:

- To implement this pattern, the designer should replicate the channel which includes the replication of the hardware as well as software.
- With respect to the sensor, there are two options either to use a common single sensor for the three channels which may become a source for a single-point-failure, or to use three separate sensors one for each channel [57], which will increase the recurring cost.
- If the outputs from the three channels are binary and identical without any deviation, then the voting can be implemented as bit-by-bit comparator that checks for a common output between two channels or more.
- If the three modules consist of different hardware components, then there might be a divergence between the correct outputs [97]. So, the designer should determine a tolerance value δ, such that the two outputs A and B will be considered as identical as long as $|A - B| < \delta$.

Consequences and Side Effects:

- The main drawback for this pattern is that it is not appropriate for systematic faults handling. In this case the three channels are identical and have the same possible fault, and the system will continue to work producing invalid data.
- To deal with single-point of failure in the input sensor, it is possible to use three separate sensors. This option might lead to a problem, especially for different sensors with different speed of responses [57].

Related Patterns:

- This pattern does not perform a check on the input data or on the actuators. So, it is possible to combine this pattern with the Single Protected Channel Pattern in order to deal with the transient fault.
- In order to deal with the systematic faults, this pattern can combined with the heterogeneous design pattern to design three diverse modules that perform the same functionality. This option will increase the development cost to 300%.

7.6. M-Out-Of-N Pattern (M-oo-N)

Other Names:
M/N Parallel Redundancy Pattern.

Type:
Hardware Pattern.

Abstract:
The M-oo-N pattern includes homogeneous hot redundancy. It consists of *N* identical modules (channels) which operate in parallel to mask random faults, and to enhance system safety and reliability. The M-oo-N redundancy requires that at least *M* components succeed out of the total *N* parallel modules for the system to succeed. An M-out-of-N voting algorithm is used in the voter component to allow system operating and providing the required functionality in the presence of random faults without losing the input data.

Context:
Developing an embedded system with/without fail-safe-state in a situation that includes high random failure rate and no limitation on redundancy, with the purpose of improving system safety and reliability.

Problem:
How to deal with random faults in order to increase the safety and reliability of the system without losing the input data.

Pattern Structure:
The pattern structure, which is shown in Fig. 7.6, contains *N* identical modules or channels in which *M* out of these *N* channels must be functioning for the system to be functioning and producing the correct actuation control signals. The voting element plays the main role in this pattern since it is used to find the possible correct result by performing the M-oo-N voting strategy. Like in TMR, this pattern does not identify the type or the reason for the fault; it just determines the modules which contain the fault without searching for reasons.

The function of each unit is as follows:

- *Module (Channel)*: (see the general pattern in Section 6.7).
- *Input Sensor(s)*: It represents the source of information for this pattern (see the general pattern in Section 6.7). It is possible to use one sensor for all the modules, but in this case the failure of this sensor would affect

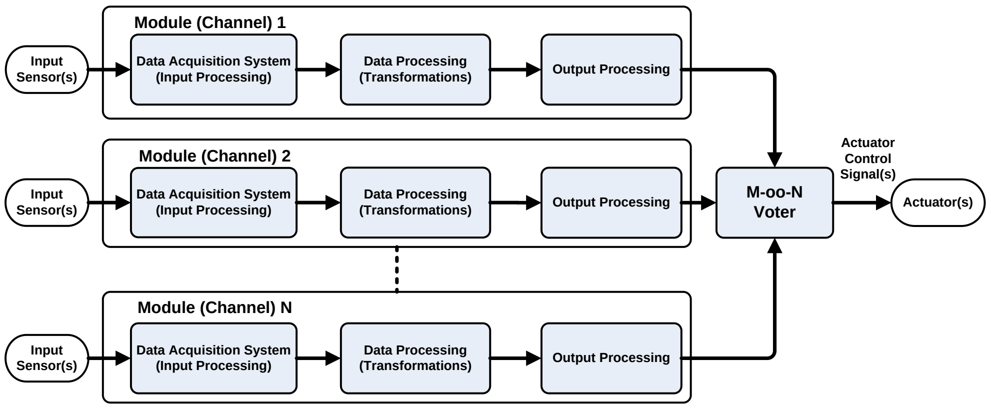

Module (Channel) 1
Input Sensor(s)
Data Acquisition System (Input Processing)
Data Processing (Transformations)
Output Processing
Module (Channel) 2
Input Sensor(s)
Data Acquisition System (Input Processing)
Data Processing (Transformations)
Output Processing
M-oo-N Voter
Actuator Control Signal(s)
Actuator(s)
Module (Channel) N
Input Sensor(s)
Data Acquisition System (Input Processing)
Data Processing (Transformations)
Output Processing

Figure 7.6.: M-out-of-N Redundancy Pattern

inputs to all of the modules. Another choice is to use N separate sensors in order to remove the possible single-point of failure.

- *Actuator(s)*: (see the general pattern in Section 6.7).
- *Data Acquisition (Input Processing)*: (see the general pattern in Section 6.7).
- *Data Processing (Transformation)*: (see the general pattern in Section 6.7).
- *Output Processing*: (see the general pattern in Section 6.7).
- *M-oo-N Voter*: The M-oo-N voter receives N outputs from the output processing units of the modules, and then it produces a valid output only if at least M channels produce an identical result, else the action of the voter depends on the nature of the system:
  – Switching the system into its fail-safe state if it exists.
  – Discarding the result and arising a flag to indicate unsafe condition.
  – Choosing an output based on a predefined policy (*not recommended*).
  Like the sensor, this component could be a reason for a single-point of failure. So, it should be carefully designed and tested.

Implication:
This section shows the implication of this pattern relative to the basic system.

- ***Reliability***:
  Assume that the voter is a simple component that was carefully designed with reliability ($R_{voter} \approx 1$).

The M-oo-N redundancy will continue to work correctly as long as at least M modules have no fault.

$$R_{new} = \sum_{i=M}^{N} \binom{N}{i} (R)^i (1-R)^{N-i} \tag{7.21}$$

$\Rightarrow$ The percentage relative improvement in reliability is

$$\begin{aligned} RRI &= \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% \\ RRI &= \frac{\left(\sum_{i=M}^{N} \binom{N}{i} (R)^i (1-R)^{N-i}\right) - R}{1-R} \times 100\% \end{aligned} \tag{7.22}$$

- ***Safety***:
  The M-oo-N redundancy pattern includes two design techniques: *test by redundant hardware*, and *fault detection and diagnosis (voter)*. According to the hardware and software requirements in the standard IEC 61508-2, 3 [46], the recommendations for these techniques are shown in Tab. 7.8.

Table 7.8.: Recommendations for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Test by redundant hardware	R	R	R	R
Fault detection and diagnosis (Voter)	–	R	HR	HR

According to the last table, the average recommendations of this pattern for the different safety integrity levels are shown in Tab. 7.9.

Table 7.9.: Recommendations of M-oo-N Pattern for safety integrity levels.

Pattern	**SIL1**	**SIL2**	**SIL3**	**SIL4**
M-oo-N Redundancy Pattern	WR	R	MR	MR

To compute RSI:

$$P_{UF(old)} = 1 - R \tag{7.23}$$

The M-oo-N redundancy pattern will lead to an unsafe failure when at least M modules contain identical faulty results. Therefore, the probability that the system will give an erroneous result that leads to unsafe failure is:

$$P_{UF(new)} = \sum_{i=M}^{N} \binom{N}{i} (1-R)^i (R)^{N-i} \tag{7.24}$$

$\Rightarrow$ The percentage relative safety improvement is

$$\begin{aligned}
RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\
RSI &= \left(1 - \frac{P_{UF(new)}}{P_{UF(old)}}\right) \times 100\% \\
RSI &= \left(1 - \frac{\sum_{i=M}^{N} \binom{N}{i} (1-R)^i (R)^{N-i}}{1-R}\right) \times 100\% \qquad (7.25)
\end{aligned}$$

Unlike TMR pattern, the safety and reliability improvement in this pattern are not equal. Different values of M achieve different combinations of reliability and safety. In the applications with moderate reliability and high safety requirement, high value of M may be preferable to be used for voting algorithm, where $(M = N)$ gives the highest possible safety with reduced reliability [98].

- ***Cost***:
  The cost of this pattern can be classified into two parts:
  - *Recurring Cost*:
    - This pattern has a high recurring cost due to the using of N parallel modules. So, the recurring cost is $N \times 100\%$ comparing to the basic system.
    - The extra cost includes the M-oo-N voter which is normally a simple hardware circuit that depends on the type of the output signals and the implementation method.
  - *Development Cost*: The used modules are identical and using the same software. Therefore, the development cost for this pattern is similar to the development cost of the basic system.
- ***Modifiability***:
  This pattern contains identical channels; thus, it does not change the level of modifiability of the basic system. The effort to modify the functionality of a system, which includes M-oo-N pattern, will be equivalent to modifying a simple channel.
- ***Impact on Execution Time***:
  Like TMR, this pattern has a little influence on the executing time comparing to the basic system, since the N modules are running separately in parallel. The only time influence of this pattern is the small delay of the voter hardware that affects the response time from reading the input signal to generating the control actuating signals.

Implementation:

- For homogeneous implementation of this pattern, the designer should use the same hardware as well as the software for all the channels.
- To overcome the single-point of failure problem in the sensor, a separate sensor can be used for each channel, which will increase the recurring cost.
- If the outputs from the channels are binary and identical without any deviation, then the voting can be implemented as bit-by-bit comparator that checks for a common output between at least M channels.
- If the hardware diversity concept is used in the implementation to solve the problem of systematic faults, then the possible deviation in value or time between the correct outputs should be taken into consideration in the design of the voting system.

Consequences and Side Effects:

- The main drawback for this pattern is that it is not appropriate for systematic faults handling because the channels are identical and have the same possible fault. Thus, the M-oo-N pattern cannot detect this type of faults, which means that the system continues producing invalid data.
- If multiple sensors are used to deal with single-point of failure in the input sensor, then this solution might lead to a problem, especially for different sensors with different speed of responses [57].

Related Patterns:

- The M-oo-N redundancy pattern has a problem with systematic faults. So, it is possible to combine this pattern with the heterogeneous design pattern to design N heterogeneous modules with the same functionality. This option will increase the development cost by N times.
- Since the hardware channels run in parallel, the concept of software diversity can also be used to solve the problem of software faults.
- For the applications with high safety requirement, the M-oo-N pattern can be used to implement the monitoring channel of the Monitor-Actuator Pattern. This combination will improve the safety monitoring of such systems by providing a high reliable monitoring channel.

7.7. Monitor-Actuator Pattern (MA)

Other Names:
–

Type:
Hardware Pattern.

Abstract:
The Monitor-Actuator Pattern [37] is a special type of heterogeneous redundancy that is suitable for safety critical systems with low availability requirements and a fail-safe state, which is a condition of the system known to be always safe [37]. The MA pattern consists of two different channels (modules): A primary channel called the actuation channel, which performs the main action such as controlling some actuators, and a monitoring channel, which provides a monitoring for the actuation channel in order to detect and to identify the possible faults and then to make the actuation channel entering its fail-safe state. The monitoring channel differs from the actuator channel such that if it contains any fault, the actuation channel continues to operate properly.

Context:
Developing an embedded system with a given fail-safe state and low availability requirements, where some level of redundancy is acceptable.

Problem:
How to improve the safety of an embedded system in the presence of a single-point of failure in a system that includes a fail-safe state and low availability requirements at reasonable cost.

Pattern Structure:
The structure of the Monitor-Actuator Pattern is shown in Fig. 7.7. It consists of two channels that run independently and in parallel to provide the required functionality and a monitoring method to switch the system into its fail-safe state in the presence of failure.

The function of each unit is as follows:

- *Actuation Channel*: (see the general pattern in Section 6.7).
- *Input Data Source*: (see the general pattern in Section 6.7).
- *Actuator(s)*: (see the general pattern in Section 6.7).

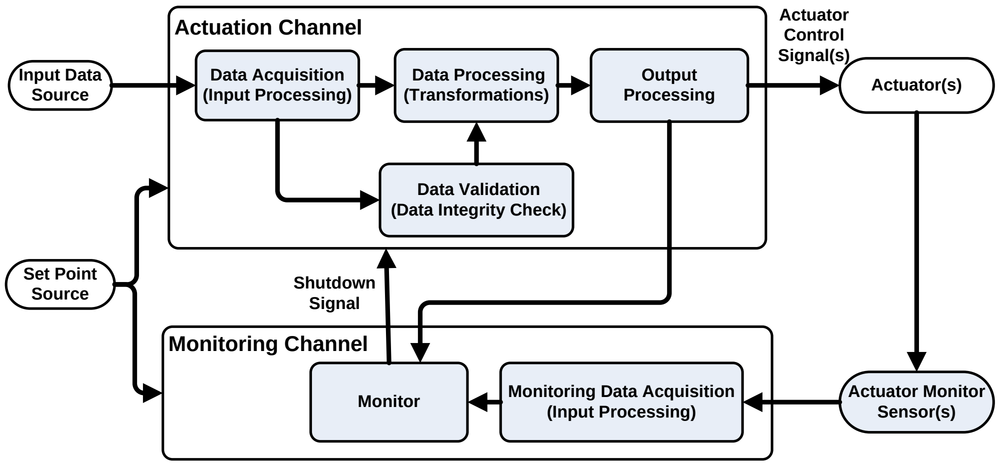

Actuation Channel
Input Data Source
Data Acquisition (Input Processing)
Data Processing (Transformations)
Output Processing
Actuator Control Signal(s)
Actuator(s)
Data Validation (Data Integrity Check)
Set Point Source
Shutdown Signal
Monitoring Channel
Monitor
Monitoring Data Acquisition (Input Processing)
Actuator Monitor Sensor(s)

Figure 7.7.: Monitor-Actuator Pattern

- *Data Acquisition (Input Processing)*: (see the general pattern in Section 6.7).
- *Data Processing (Transformation)*: (see the general pattern in Section 6.7).
- *Output Processing*: (see the general pattern in Section 6.7).
- *Data Validation (Data Integrity Check)*: It provides a check on the input data and the system itself during the executing of the desired algorithm to ensure that the internal processing is proceeding properly.
- *Actuator Monitor Sensor(s)*: It represents the source of information to the monitoring channel, where it is used to get feedback signals from the actuators in order to provide a monitoring for the actuation channel.
- *Set Point Source*: It can be considered as the source of commanded actuation control signals, which supplies the set point for the actuation channel. The set point signals should be provided to the two channels.
- *Monitoring Channel*: It is used to improve the safety of the system by providing a continuous monitoring for the actuation channel to check its proper operation. It takes the information from the set point source and the actuator sensors to detect possible faults in the actuation channel. In the case of improper operation, it forces the actuation channel to enter the fail-safe state. The monitoring channel is independent from the actuation channel, so if there is a fault in the monitoring channel, then the actuation channel will continue to operate properly, but in this case any fault in the actuation channel can not be detected.

- *Monitoring Data Acquisition*: It collects the raw data from the actuator sensors and may convert the data to another form as in analog to digital converter (ADC). Finally, it transfers the data to the monitor component.
- *Monitor*: The monitor takes the information about the outputs of the actuators, which is collected by the actuator sensors and processed by the monitoring acquisition system, and compares it with the provided set points to make sure that the actuation channel is working properly. If the result of the comparison shows improper operation in the actuation channel, the monitor generates a shutdown signal to the actuation channel.

Implication:
This section shows the implication of this pattern relative to the basic system.

- ***Reliability***:
  As shown previously, this pattern is used to detect and to identify the faults in the actuation channel, then to generate the shutdown signal to force the actuation channel entering its fail-safe state. On the other hand, this pattern does not change the failure rate of the system ($\lambda_{new} = \lambda_{old}$), and the actuation channel cannot continue to function when a fault is detected. Thus, this pattern does not improve the reliability of the system.

  $$R_{new} = R_{old} \tag{7.26}$$

  $\Rightarrow$ The percentage relative improvement in reliability is

  $$RRI = \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% = 0\% \tag{7.27}$$

- ***Safety***:
  The Monitor-Actuator Pattern includes two design techniques: *test by redundant hardware* and *safety bag techniques*. According to the hardware and software requirements in the standard IEC 61508-2, 3 [46], the recommendations for these techniques are shown in Tab. 7.10.

Table 7.10.: Recommendations for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Test by redundant hardware	R	R	R	R
Safety bag techniques	–	R	R	R

According to the last table, the average recommendations of this pattern for the different safety integrity levels are shown in Tab. 7.11.

Table 7.11.: Recommendations of MA Pattern for safety integrity levels.

Pattern	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Monitor-Actuator Pattern	WR	R	R	R

To compute RSI, we will use the following notations for this pattern:

– R_{AC} :	The reliability of the actuation channel which is equal to a simple channel ($R_{AC} = R$).
– R_{MC} :	The reliability of the monitoring channel.
– C_{MC} :	The coverage factor of the monitoring channel which is equal to the probability that a failure in the actuation channel is detected by the monitoring channel.

The probability of failure in the actuation channel is equal to the failure in a simple channel

$$P_{UF(old)} = 1 - R_{AC} = 1 - R \tag{7.28}$$

The failures in the actuation channel can be divided into two groups:

1. Safe failures detected by the monitoring channel.
2. Unsafe failures that are undetected by the monitoring channel due to either a failure in the monitoring channel or uncovered failure.

The probability of safe failure is

$$P_{SF} = (1 - R_{AC})\, R_{MC}C_{MC} = (1 - R)\, R_{MC}C_{MC} \tag{7.29}$$

The probability of unsafe failure can be calculated as follows:

$$\begin{aligned} P_{UF(new)} &= (1 - R) - (1 - R)\, R_{MC}C_{MC} \\ P_{UF(new)} &= (1 - R)\,(1 - R_{MC}C_{MC}) \qquad (7.30) \end{aligned}$$

⇒ The percentage relative safety improvement is

$$\begin{aligned} RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\ RSI &= \frac{(1 - R)\,(1 - R_{MC}C_{MC}) - (1 - R)}{0 - (1 - R)} \times 100\% \\ RSI &= R_{MC}C_{MC} \times 100\% \qquad (7.31) \end{aligned}$$

It is clear from Equation (7.31) that the safety improvement in this pattern depends on the *reliability* and *coverage factor* of the monitoring channel.

- ***Cost***:
  The cost of this pattern can be classified into two parts:
  - *Recurring Cost*: The extra cost includes
    - The cost of extra sensors (actuator sensors).
    - The cost of the monitoring channel which contains two components: the data acquisition and the monitor. Comparing to the basic channel, this new channel is simple and less expensive.
  - *Development Cost*: It includes the cost for developing the monitoring channel.

  In general, this pattern is less expensive comparing to the other patterns that include high level of redundancy.

- ***Modifiability***:
  It is not difficult to modify this pattern by adding extra functionality to the actuation channel. This step involves the need to modify the monitoring channel to provide a monitoring to the new features, which will include additional sensors and the modification of the monitoring data acquisition and the *Monitor* component to take the new features into consideration.

- ***Impact on Execution Time***:
  The two channels in this pattern run simultaneously in parallel; thus there is no effect for the monitoring channel on the actuation during the normal operation of the system. Even if there is a fault in the monitoring channel, the actuation channel will not be affected. Therefore, there is no influence on the execution time of the actuation channel.

Implementation:

- In the implementation of the monitoring channel, many factors should be taken into consideration such as the time lag, measurement accuracy, computational accuracies, data formatting errors, and many other factors that might affect the comparison process.

- It is necessary to distinguish between transient and persistent faults, because in some situation a single transient fault may not be harmful and can be neglected, but this is not the case for persistent fault that should be detected by the monitoring channel. So, it is a good idea for the monitoring channel to store some historical information about the monitored value which could be helpful to determine whether the detected value represents a transient or persistent fault.

- As shown previously, if there is any fault in the monitoring channel, then the system can operate properly, but there will be a problem if a second fault occurs in the actuation channel. Thus, the monitoring channel must be checked to make sure that the system meets the required safety function. The check can be one or a combination of the following :
  1. Power-On Self Tests (*POSTs*).
  2. Periodic Built-In Tests (*BITs*).
  3. A life tick message exchanges between the actuator and monitoring channel.

  If the used check fails, then this indicates that a fault exists in the system, and the fail-safe state is entered.
- The previous checks must be preformed in a time interval less than the mean-time between failures (*MTBF*) of the monitoring channel and any of its components [37].

Consequences and Side Effects:
The main drawback for this pattern is that it is not appropriate for applications with high availability requirements.

Related Patterns:
The safety of the actuation channel depends on the reliability of the monitoring channel that generates the shutdown signal in the presence of a fault in the actuation channel. So, it is possible to use more complex patterns for the monitoring channel such as homogeneous duplex, heterogeneous duplex, triple modular redundancy, and M-oo-N redundancy pattern.

7.8. Sanity Check Pattern (SC)

Other Names:
–

Type:
Hardware Pattern.

Abstract:
The Sanity Check Pattern [37] is considered as a lightweight pattern which is used to ensure that the basic channel is approximately correct. It is a variant of the Monitor-Actuator Pattern which is suitable for a situation that has low availability requirements, a fail-safe state, and when the fine control of the system does not affect the system safety [37]. Similar to the Monitor-Actuator Pattern, it consists of two different channels (modules): An actuation channel, which performs the action such as controlling some actuators, and a sanity channel, which provides a monitoring to the actuation channel to ensure that the actuation output is approximately correct and within some fixed range. The sanity channel is different from the actuation channel and it may include lower cost and lower accuracy sensors used to identify faults that include a large deviation in the actuator output from the commanded set point.

Context:
Developing an embedded system with a fail-safe state and low availability requirements where fine control is not a safety property of the actuation channel.

Problem:
How to improve the safety of an embedded system in the presence of single-point of failure in a system that includes a fail-safe state and low availability requirement. This involves how to continue providing the required safety level to ensure that the system does no injure or harm when there is any deviation in the output of the actuators from the commanded set point at a low and reasonable cost.

Pattern Structure:
The Sanity Check Pattern is derived from the Monitor-Actuator Pattern since they have identical structures. The pattern structure is shown in Fig. 7.8. It consists of two channels that run independently and in parallel to provide the required functionality, and a safety monitoring method to switch the system into its fail-safe state in the presence of failure.

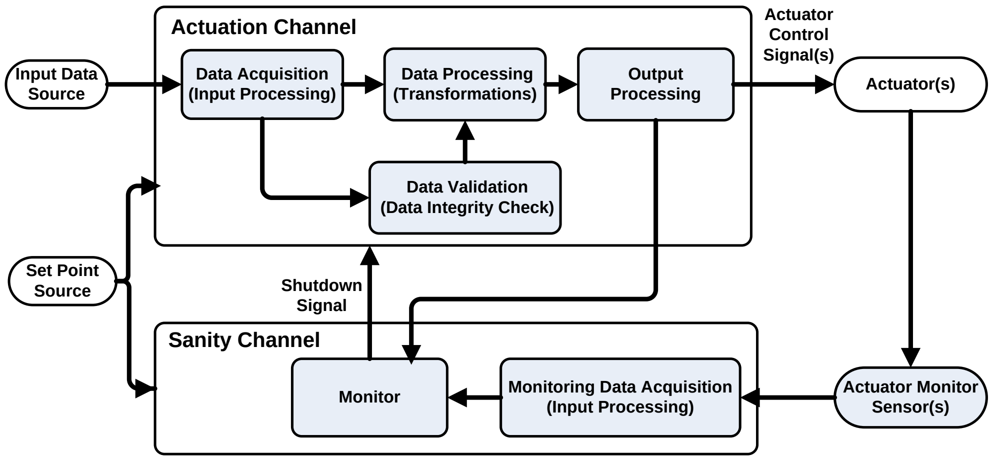

Actuation Channel
Input Data Source
Data Acquisition (Input Processing)
Data Processing (Transformations)
Output Processing
Actuator Control Signal(s)
Actuator(s)
Data Validation (Data Integrity Check)
Set Point Source
Shutdown Signal
Sanity Channel
Monitor
Monitoring Data Acquisition (Input Processing)
Actuator Monitor Sensor(s)

Figure 7.8.: Sanity Check Pattern

The function of each unit is as follows:

- *Actuation Channel*: (see the general pattern in Section 6.7).
- *Input Data Source*: (see the general pattern in Section 6.7).
- *Actuator(s)*: (see the general pattern in Section 6.7).
- *Data Acquisition (Input Processing)*: (see the general pattern in Section 6.7).
- *Data Processing (Transformation)*: (see the general pattern in Section 6.7).
- *Output Processing*: (see the general pattern in Section 6.7).
- *Data Validation (Data Integrity Check)*: It provides a check on the input data and the system itself during the executing of the desired algorithm to ensure that the internal processing is proceeding properly.
- *Actuator Monitor Sensor(s)*: It represents the source of information to the sanity channel, where it is used to get feedback signals from the actuators in order to provide a monitoring for the actuation channel.
- *Set Point Source*: It can be considered as the source of commanded actuation control signals, which supplies the set point for the actuation channel. The set point signals should be provided to the two channels.
- *Monitoring Channel*: This channel is identical to the *monitoring channel* in the Monitor-Actuator Pattern, where it is used to check on the correct

operation of the actuation channel. It takes information from the set point source and the actuator sensors to detect possible faults in the actuation channel. In the case of *great difference* between the set point and the measured value, the sanity channel forces the actuation channel entering the fail-safe state. The sanity channel is independent from the actuation channel, so if there is a fault in the sanity channel, the actuation channel continues to operate properly, but in this case any further fault in the actuation channel cannot be detected.

- *Monitoring Data Acquisition*: It collects the raw data from the actuator sensors and may convert the data to another form as in analog to digital converter (ADC). Finally, it transfers the data to the monitor component.
- *Monitor*: The monitor takes the information about the outputs of the actuators, which is collected by the actuator sensors and processed by the monitoring acquisition system, and compares it roughly with the provided set point commands to ensure that the actuation output is approximately correct and within a given range. If the result of the comparison shows that the output is totally incorrect and may affect the safety of the system, the monitor generates a shutdown signal to the actuation channel. The comparison process in this unit is very simple and less accurate than in the Monitor-Actuator Pattern. So, this component is simpler and less expensive than the monitor unit in the Monitor-Actuator Pattern.

Implication:
This section shows the implication of this pattern relative to the basic system.

- ***Reliability***:
  Like in MA pattern, this pattern is used to detect and to identify the faults in the actuation channel, then to generate the shutdown signal to force the actuation channel entering its fail-safe state. On the other hand, this pattern does not change the failure rate of the system ($\lambda_{new} = \lambda_{old}$), and the actuation channel cannot continue to function when a fault is detected. Thus, this pattern does not improve the reliability of the system.

$$R_{new} = R_{old} \tag{7.32}$$

$\Rightarrow$ The percentage relative improvement in reliability is

$$RRI = \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% = 0\% \tag{7.33}$$

- ***Safety***:
  The Sanity Check Pattern includes two design techniques: *test by redundant hardware* and *safety bag techniques*. According to the hardware and software requirements in the standard IEC 61508-2, 3 [46], the recommendations for these techniques are shown in Tab. 7.12.

Table 7.12.: Recommendations for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Test by redundant hardware	R	R	R	R
Safety bag techniques	–	R	R	R

According to the last table, the average recommendations of this pattern for the different safety integrity levels are shown in Tab. 7.13.

Table 7.13.: Recommendations of SC Pattern for safety integrity levels.

Pattern	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Sanity Check Pattern	WR	R	R	R

To compute RSI, we will use the following notations for this pattern:

– R_{AC} :	The reliability of the actuation channel which is equal to a simple channel ($R_{AC} = R$).
– R_{SnC} :	The reliability of the sanity channel.
– C_{SnC} :	The coverage factor of the sanity channel which is equal to the probability that a failure in the actuation channel is detected by the sanity channel.

The probability of failure in the actuation channel is equal to the failure in a simple channel

$$P_{UF(old)} = 1 - R_{AC} = 1 - R \tag{7.34}$$

The failures in the actuation channel can be divided into two groups:

1. Safe failures detected by the sanity channel.
2. Unsafe failures that are undetected by the sanity channel due to either a failure in the sanity channel or uncovered failure.

The probability of safe failure is

$$P_{SF} = (1 - R_{AC})\, R_{SnC} C_{SnC} = (1 - R)\, R_{SnC} C_{SnC} \tag{7.35}$$

The probability of unsafe failure can be calculated as follows:

$$\begin{aligned} P_{UF(new)} &= (1 - R) - (1 - R)\, R_{SnC} C_{SnC} \\ P_{UF(new)} &= (1 - R)\,(1 - R_{SnC} C_{SnC}) \end{aligned} \tag{7.36}$$

$\Rightarrow$ The percentage relative safety improvement is

$$\begin{aligned} RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\ RSI &= \frac{(1 - R)\,(1 - R_{SnC} C_{SnC}) - (1 - R)}{0 - (1 - R)} \times 100\% \\ RSI &= R_{SnC} C_{SnC} \times 100\% \end{aligned} \tag{7.37}$$

The safety improvement in this pattern depends on the *reliability* and *coverage factor* of the sanity channel which is normally simpler than monitoring channel with simpler comparison algorithm. Thus, the coverage factor in this pattern is less than in the Monitor-Actuator Pattern, and gives lower safety improvement.

- ***Cost***:
  The cost of this pattern can be classified into two parts:
  - *Recurring Cost*: The extra cost includes
    * The cost of extra low-accuracy (low-cost) actuator sensors.
    * The cost of the sanity channel which includes two components: the data acquisition and the monitor. This sanity channel is very simple and it has a low recurring cost comparing to the basic channel.
  - *Development Cost*: It includes the low cost for developing a simple sanity channel that performs a simple comparison to verify that the output of the actuator does not deviate from the input set point.

  In general, this pattern is very inexpensive comparing to the other patterns which include high level of redundancy.

- ***Modifiability***:
  It is very simple to modify this pattern by adding extra functionality to the actuation channel. This step is approximately similar to modifying a

basic channel with relatively little extra work. This extra work involves the need to add more actuator sensors and to modify the comparison process of the monitor to cover the new features.

- ***Impact on Execution Time***:
  The two channels run simultaneously in parallel and there is no effect for the sanity channel on the actuation channel during the normal operation of the system. Even if there is fault in the sanity channel, the actuation channel will not be affected. Therefore, there is no influence on the time of execution for the actuation channel.

Implementation:

- To implement this pattern, two different types of sensors will be used: high sensitivity sensors for the actuation channel, and low-cost and low-sensitivity sensors for the sanity channel.
- The implementation of the monitor component is very simple since it is a very simple unit that includes a simple algorithm to perform the required broad range comparison.
- Unlike the original Monitor-Actuator Pattern, it is not necessary to take into consideration the time lag, measurement accuracy, computational accuracies, and data formatting errors in the implementation of this pattern because this pattern just includes a simple approximate comparison.

Consequences and Side Effects:
The main drawback for this pattern is that it has low coverage, which makes it not suitable for applications with high availability requirements.

Related Patterns:
It is possible to use the Monitor-Actuator Pattern for more accurate check on the actuator channel, but this will increase the design and recurring cost. In the case where no fail-safe state exists, more complex pattern may be used such as homogeneous and heterogeneous duplex pattern.

7.9. Watchdog Pattern (WD)

Other Names:
Watchdog Timer, Watchdog Processor, Hardware Watchdog Pattern [78].

Type:
Hardware Pattern.

Abstract:
The Watchdog Pattern [37] is a very lightweight and inexpensive pattern with minimal coverage that is used to check the internal computational execution of the actuation channel. It is widely used in the embedded systems to make sure that the time-dependent computational processing is proceeding properly as expected in a predefined order [37]. This pattern includes a component called a watchdog that receives periodic messages from the watched channel. If an event occurs too late or out of order, then the watchdog issues a shutdown signal or a corrective action in order to avoid losing control of the system. The watchdog pattern is rarely used alone in safety-critical systems, and it is normally used with other patterns to improve the system safety.

Context:
Developing a real time embedded system with a given fail-safe state or a corrective action and low availability requirements.

Problem:
This pattern is used to solve the following problems:

- How to add additional safety to the system by providing a simple and low-cost way to watch, detect and handle faults during the execution.
- How to make sure that the internal computational processing of the actuation channel is proceeding properly as expected.
- How to identify time-based faults.
- How to improve deadlock detection using “*Keyed Watchdog*”.

Pattern Structure:
The Watchdog Pattern is a very simple variant of the Sanity Check Pattern. While it has an actuation channel to do end-to-end actuation, it differs in the sanity channel which is replaced in this pattern with the watchdog component. The watchdog receives liveness messages (Strokes) from the actuation channel on a periodic or in a predefined-sequence base. The watchdog must be stroked within a specified period of time or it will initiate a corrective action such as a

shutdown signal. In other words, it checks the sequence or the timelines of the strokes to detect possible faults in the actuation channel. The structure of this pattern is shown in Fig. 7.9, where the function of each unit is as follows:

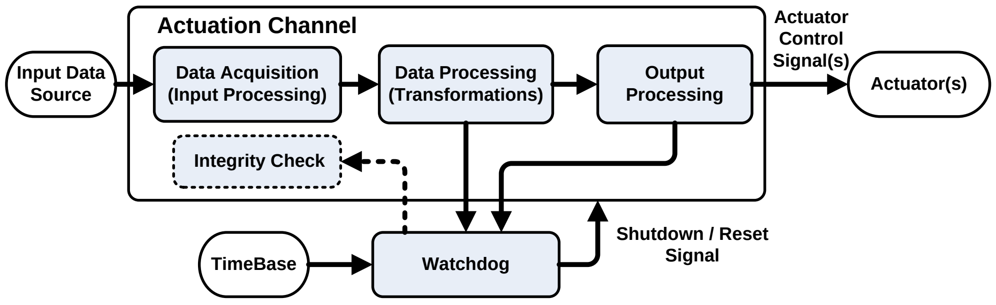


Figure 7.9.: Watchdog Pattern

- *Actuation Channel*: (see the general pattern in Section 6.7).
- *Input Data Source*: (see the general pattern in Section 6.7).
- *Actuator(s)*: (see the general pattern in Section 6.7).
- *Data Acquisition (Input Processing)*: (see the general pattern in Section 6.7).
- *Data Processing (Transformation)*: (see the general pattern in Section 6.7).
- *Output Processing*: (see the general pattern in Section 6.7).
- *Integrity Check (Optional)*: It is an optional component, which is invoked by the watchdog to run a periodic Built In Test (*BIT*) to verify all or a portion of the internal functionality of the actuation channel.
- *Time Base*: This is an independent timing source (timing circuit) that is used to drive the watchdog. This time source is separate from the one used to drive the actuation channel.
- *Watchdog*: It is the main component in this pattern which is used to observe the behavior of the actuation channel. The watchdog receives liveness messages (strokes) from the components of the actuation channel in a predefined timeframe. If a stroke comes too late or out of sequence, then the watchdog considers this situation as a fault in the actuation channel. Consequently, it issues a shutdown or reset signal to the actuation channel or initiates a corrective action through sending a command signal to the optional integrity check.

Implication:
This section shows the implication of this pattern relative to the basic system.

- ***Reliability***:
  The reliability improvement of the watchdog pattern is very small and depends on the response-action to the timeout event. If there is a corrective action, then the reliability improvement depends on the coverage of the corrective action. But if the watchdog just initiates a shutdown signal to force the actuation channel entering its fail-safe state, then this pattern will not change the failure rate of the system ($\lambda_{new} = \lambda_{old}$), and the actuation channel cannot continue to function when a fault is detected. Therefore, this pattern does not improve the reliability of the system.

  $$R_{new} = R_{old} \tag{7.38}$$

  $\Rightarrow$ The percentage relative improvement in reliability is

  $$RRI = \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% = 0\% \tag{7.39}$$

- ***Safety***:
  The watchdog pattern includes a single design technique, which is *program sequence monitoring.* According to the hardware requirements in the standard IEC 61508-2 [46], the recommendations for this technique are shown in Tab. 7.14.

Table 7.14.: Recommendations of WD Pattern for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Program sequence monitoring	HR	HR	HR	HR

To compute RSI, we will use the following notations for this pattern:

– R_{AC} :	The reliability of the actuation channel which is equal to a simple channel ($R_{AC} = R$).
– R_{WD} :	The reliability of the watchdog component.
– C_{WD} :	The coverage factor of the watchdog component which is equal to the probability that a failure in the actuation channel is detected by the sanity watchdog.

The probability of failure in the actuation channel is equal to the failure in a simple channel

$$P_{UF(old)} = 1 - R_{AC} = 1 - R \tag{7.40}$$

The failures in the actuation channel can be divided into two groups:

1. Safe failures detected by the watchdog.
2. Unsafe failures that are undetected by the watchdog due to either a failure in the watchdog or the minimal coverage of the watchdog.

The probability of safe failure is

$$P_{SF} = (1 - R_{AC})\, R_{WD}C_{WD} = (1 - R)\, R_{WD}C_{WD} \tag{7.41}$$

The probability of unsafe failure can be calculated as follows:

$$\begin{aligned} P_{UF(new)} &= (1 - R) - (1 - R)\, R_{WD}C_{WD} \\ P_{UF(new)} &= (1 - R)\,(1 - R_{WD}C_{WD}) \end{aligned} \tag{7.42}$$

$\Rightarrow$ The percentage relative safety improvement is

$$\begin{aligned} RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\ RSI &= \frac{(1 - R)\,(1 - R_{WD}C_{WD}) - (1 - R)}{0 - (1 - R)} \times 100\% \\ RSI &= R_{WD}C_{WD} \times 100\% \end{aligned} \tag{7.43}$$

If we assume that the watchdog was carefully designed ($R_{WD} \approx 1$) then

$$RSI \approx C_{WD} \times 100\% \tag{7.44}$$

Equation (7.44) shows that the safety improvement in this pattern depends on the coverage of the watchdog which normally includes the minimal coverage. For this reason, the safety improvement is less than in the Sanity Check Pattern.

- ***Cost***:
  The cost of this pattern can be classified into two parts:
  - *Recurring Cost*: The extra cost includes the cost of the watchdog, which is a simple component, and the independent timing source.
  - *Development Cost*: This pattern has the lowest development cost which depends on the recovery action initiated by the watchdog.

  In general, this pattern is inexpensive with low cost comparing to the other patterns that include high level of redundancy.

- ***Modifiability***:
  It is very simple to modify this pattern by adding extra functionality.The only things that should be done, is to know whether the new components need to send stroke messages to the watchdog or not.

- ***Impact on Execution Time***:
  Since the actuation channel and the watchdog run simultaneously in parallel, there is no effect for the watchdog on the actuation channel during the normal operation of the system except the execution of the built in tests that may be initiated by the watchdog when it is stroked.

Implementation:

- The watchdog component is simple and often implemented as a separate hardware to protect the system from software faults.
- To provide protection from time-based faults, a separate timing source such as crystal or an *R-C* circuit must be used.
- There is a problem for the watchdog that is called *Live-Lock* problem which occurs when the actuation channel gets stuck in loop that strokes the watchdog without performing the appropriate actuation. To avoid this problem, the watchdog may require a specific data in a predefined-sequence to occur with the strokes which is called *Keyed Watchdog*.
- The best approach to implement a Keyed Watchdog is to have the keys dynamically generated instead of storing them in a memory [37].
- To increase the fault coverage, it is common to invoke a *BIT*, *CRC*, or stack overflow check when the watchdog is stroked to ensure that the computational processing of the actuation channel is proceeding properly.

Consequences and Side Effects:
The main drawback for this pattern is the minimal coverage of the watchdog; thus, it is rarely used alone in safety critical systems. It may be combined with other safety patterns.

Related Patterns:
The Watchdog Pattern is a very lightweight pattern that is rarely used alone in safety critical system. It is normally associated with other patterns as a method to identify time-based faults and to drive periodic BITs. In the case where no fail-safe state exists, more complex pattern may be used such as homogeneous and heterogeneous duplex pattern.

7.10. Safety Executive Pattern (SE)

Other Names:
Safety Kernel Pattern [36].

Type:
Hardware Pattern.

Abstract:
The Safety Executive Pattern [37] is a large scale pattern that is suitable for complex and highly safety-critical systems. It is a smart extension of the Watchdog Pattern targeting the problem where a shutdown of the system by the actuation channel itself might be critical or take too long time. This problem occurs especially in those systems in which a complicated series of steps involving several components is necessary to reach a fail-safe state. Therefore, the Safety Executive Pattern uses a watchdog in a combination with an additional safety executive component, which is responsible for the shutdown of the system as soon as the watchdog sends a shutdown signal. If the system has a fail-safe state, the safety executive component controls the shutdown of the actuation channels. Otherwise, the safety executive component has to conduct actions required to delegate an additional fail-safe processing channel.

Context:
Developing a highly safety-critical system in a situation where:

- The actuation channel requires a risk reduction by safety measures.
- The considered system has at least one fail-safe state, or an additional fail-safe processing channel has to be used to overtake necessary actions.
- The shutdown of the actuation channel is a complex process, where several safety-related actions have to be controlled simultaneously.
- A sufficient determination of failures in the actuation channel can be achieved by a watchdog.

Problem:
How to provide a centralized and consistent method for monitoring and controlling the execution of a complex safety measure in case of failures.

Pattern Structure:
The Safety Executive Pattern is based on an actuation channel to perform the required functionality and an optional fail-safe processing channel that is dedicated to the execution and control of the fail-safe processing. The central part

of this pattern is the existence of a centralized safety executive component coordinating all safety-measures required to shut down the system or to switch over to the fail-safe processing channel. The safety executive component can also be used to control multiple actuation channels in the system. The structure of this pattern is shown in Fig. 7.10, where the function of each unit is as follows:

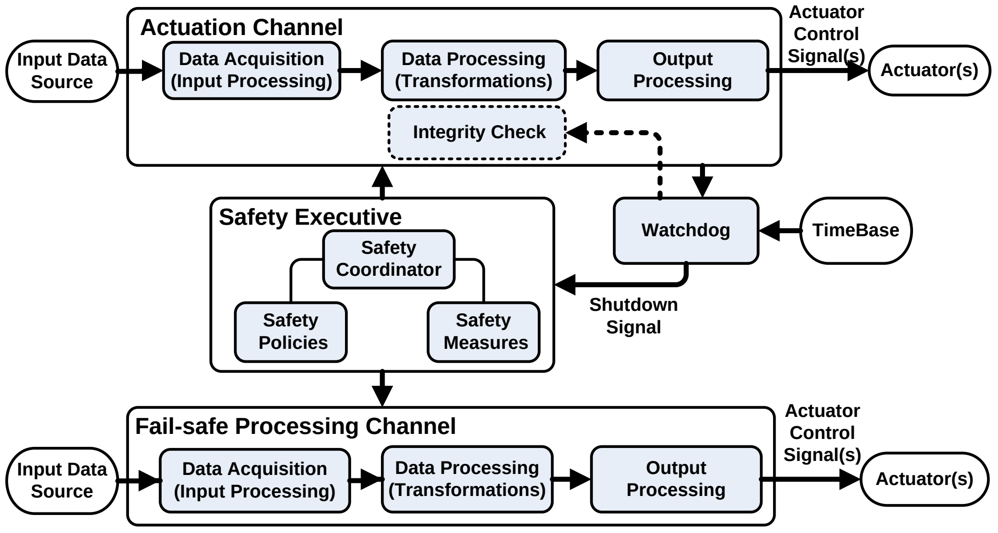


Figure 7.10.: Safety Executive Pattern

- *Actuation Channel*: (see the general pattern in Section 6.7).
- *Input Data Source*: (see the general pattern in Section 6.7).
- *Actuator(s)*: (see the general pattern in Section 6.7).
- *Data Acquisition (Input Processing)*: (see the general pattern in Section 6.7).
- *Data Processing (Transformation)*: (see the general pattern in Section 6.7).
- *Output Processing*: (see the general pattern in Section 6.7).
- *Fail-Safe Processing Channel*: It is an optional channel dedicated to the execution and control of the fail-safe processing. In the presence of a fault in the actuation channel, the safety executive turns off the actuation channel, and the fail-safe processing channel takes over. If the system does not have a fail-safe channel, then the actuation channels must have at least one reachable fail-safe state.

- *Integrity Check (Optional)*: It is an optional component, which is invoked by the watchdog to run a periodic Built In Test (*BIT*) to verify all or a portion of the internal functionality of the actuation channel.
- *Time Base*: This is an independent timing source (timing circuit) that is used to drive the watchdog. This time source is separate from the one used to drive the actuation channel.
- *Watchdog*: The watchdog receives liveness messages (strokes) from the components of the actuation channel in a predefined timeframe. If a stroke comes too late or out of sequence, then the watchdog considers this situation as a fault in the actuation channel. Consequently, it issues a shutdown signal to the safety executive component or initiates a corrective action through sending a command signal to the optional integrity check. If the system contains multiple actuation channels, then it may contain multiple watchdogs, one per actuation channel.
- *Safety Executive*: This is the main component in the Safety Executive Pattern. It tracks and coordinates all safety monitoring to ensure the execution of safety actions. It contains a safety coordinator that controls safety measures and safety policies. In the case of failure in the actuation channel, the safety executive component captures the shutdown signal from the watchdog to initiate the safety processing.
- *Safety Coordinator*: It is used to control and coordinate the safety processing that is managed by the safety measures. It also executes the control algorithms that are specified by the safety policies.
- *Safety Measures*: Include the detailed description of the safety measures. The safety coordinator may control multiple safety measures.
- *Safety Policies*: Each policy specifies a strategy or control algorithm for the safety coordinator. It involves a sequence of steps with multiple safety measures. The separation of the coordinator from the safety policies facilitates the modification and adaptation of a safety policy.

Implication:
This section shows the implication of this pattern relative to the basic system.

- ***Reliability***:
  We will use the following notations for this pattern:

– R_{AC}: The reliability of the actuation channel ($R_{AC} = R$).

– R_{SC}: The reliability of the fail-safe processing channel.

– R_{SE}: The reliability of the safety executive component.

– C_{SE}: The coverage factor which is defined as the probability that a fault in an actuation channel will be identified by the watchdog and the safety executive, and the fail-safe processing channel will be activated.

Assume that the watchdogs are carefully designed with ($R_{WD} \approx 1$). The safety executive pattern will continue to work without system failure as long as one of the following two conditions holds:

- There is no fault in the actuation channel.
- There is a fault in the actuation channel and the safety executive detects this fault and activates the fail-safe processing channel.

$$R_{new} = R_{AC} + C_{SE}R_{SE}\left(1 - R_{AC}\right)R_{SC} \tag{7.45}$$

$\Rightarrow$ The percentage relative improvement in reliability is

$$\begin{aligned} RRI &= \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% \\ RRI &= \frac{R_{AC} + C_{SE}R_{SE}\left(1 - R_{AC}\right)R_{SC} - R_{AC}}{1 - R_{AC}} \times 100\% \\ RRI &= C_{SE}R_{SE}R_{SC} \times 100\% \end{aligned} \tag{7.46}$$

- ***Safety***:
  The Safety Executive Pattern includes the following four design techniques: *program sequence monitoring with a watchdog*, *test by redundant hardware*, *safety bag techniques*, and *graceful degradation*. According to the hardware and software requirements in the standard IEC 61508-2, 3 [46], the recommendations for this technique are shown in Tab. 7.15.

Table 7.15.: Recommendations for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Program sequence monitoring (WD)	HR	HR	HR	HR
Test by redundant hardware	R	R	R	R
Safety bag techniques	–	R	R	R
Graceful degradation	R	R	HR	HR

According to the last table, the average recommendations of this pattern for the different safety integrity levels are shown in Tab. 7.16.

The probability of failure in the actuation channel is equal to the failure in a simple channel

$$P_{UF(old)} = 1 - R_{AC} = 1 - R \tag{7.47}$$

Table 7.16.: Recommendations of SE Pattern for safety integrity levels.

Pattern	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Safety Executive Pattern	WR	R	R	R

The failures in the actuation channel can be divided into two groups:

1. Safe failures identified by the safety executive component and handled by activating the fail-safe processing channel.
2. Unsafe failures, those are unhandled due to a failure in the safety executive component, a failure in the fail-safe processing channel, or uncovered failure.

The probability of safe failure is

$$P_{SF} = (1 - R_{AC})\, R_{SE}C_{SE}R_{SC} = (1 - R)\, R_{SE}C_{SE}R_{SC} \tag{7.48}$$

The probability of unsafe failure can be calculated as follows:

$$\begin{aligned} P_{UF(new)} &= (1 - R) - (1 - R)\, R_{SE}C_{SE}R_{SC} \\ P_{UF(new)} &= (1 - R)\,(1 - R_{SE}C_{SE}R_{SC}) \end{aligned} \tag{7.49}$$

$\Rightarrow$ The percentage relative safety improvement is

$$\begin{aligned} RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\ RSI &= \frac{(1 - R)\,(1 - R_{SE}C_{SE}R_{SC}) - (1 - R)}{0 - (1 - R)} \times 100\% \\ RSI &= R_{SE}C_{SE}R_{SC} \times 100\% \end{aligned} \tag{7.50}$$

Equation (7.50) shows that the safety improvement in this pattern depends on the reliability and coverage factor of safety executive component as well as on the reliability of the fail-safe processing channel.

- ***Cost***:
  This pattern is an expensive pattern with very high cost.
  - *Recurring Cost*: The extra cost includes the cost of the the actuation channel, fail-safe processing channel, the safety executive component, the watchdogs and their independent timing source.
  - *Development Cost*: The development cost for this pattern is very high since it includes a development of three different systems (channels) that include different architectures and different designs.

- ***Modifiability***:
  There are two type of possible modifications:
  - To modify the actuation channel, the designer should determine whether the new components need to send stroke messages to the watchdog or not, which can be done easily.
  - One of the main features of this pattern is the centralized safety processing.The safety executive separates the coordinator from the safety policies to simplify the modification of the safety policy.
- ***Impact on Execution Time***:
  The actuation channel and the safety executive have different CPUs, different memories and run simultaneously in parallel. Therefore, there is no effect for the safety executive component on the actuation channel, during the normal operation, except the execution of the periodic built in tests.

Implementation:

- The actuation channel, safety executive, and the fail-safe processing channel should run separately in parallel on different CPUs and memories.
- The safety-critical information must be protected against data corruption, e.g. by using CRCs or any other method to detect data errors.
- The watchdog component is simple and often implemented as a separate hardware device. It is capable of detecting a variety of hardware and software fault. However, its actual diagnostic coverage depends on the integrity check implemented in the actuation channel.
- To provide protection from time-based faults, separate timing sources must be used for the watchdog, safety executive, and the two channels.

Consequences and Side Effects:
The main drawback of this pattern is the high complexity for implementation. Therefore, it is used for complex and highly safety-critical systems.

Related Patterns:
The Safety Executive Pattern is used for complex safety-critical applications and it covers a large set of features such as sequence monitoring provided by watchdog and switch-to-backup as in the fail-safe channel. For simpler systems with simpler safety requirements, other simpler patterns such as Watchdog Pattern, Sanity Check Pattern and Monitor-Actuator Pattern can be used.

8. Software Patterns

8.1. Notations

In addition to the notations presented in Section 5.5.3 and Section 7.1, Table 8.1 shows the notations that will be used in this chapter for software patterns.

Table 8.1.: Notations for software patterns.

f	Probability that a software version will produce a failure due to a bug in its implementation
R_i	Reliability of a software version i. ($R_i = 1 - f$)
E	Event that the output of a version is erroneous. ($P\{E\} = f$)
T	Event that the acceptance test reports that the output is wrong
P_{TP}	Probability that a version will pass the acceptance test, given that the outcome is correct. ($P_{TP} = P\{\overline{T}\|\overline{E}\}$)
P_{FP}	Probability that a version will pass the acceptance test, given that the outcome is wrong. ($P_{FP} = P\{\overline{T}\|E\} = 1 - P_{TN}$)
P_{TN}	Probability that a version will fail the acceptance test, given that the outcome is wrong. ($P_{TN} = P\{T\|E\}$)
P_{FN}	Probability that a version will fail the acceptance test, given that the outcome is correct. ($P_{FN} = P\{T\|\overline{E}\} = 1 - P_{TP}$)
N	Number of available independent versions
M	Agreement number which is equal to $\lceil (N+1)/2 \rceil$ for the majority voting

8.2. Implementation Issues for Software Diversity

Since the pattern presented in this chapter share the concept of diversity programming to generate N-Version software, the following general implementation issues should be taken into consideration as long as possible during the development process in order to increase the level of diversity and the independence of the designed versions:

- The use of complete, correct, and carefully documented specifications to prevent an error in specifications from propagating to the versions.
- The use of independent and isolated teams of programmers with diversity in their training and experience.
- The use of diverse algorithms.
- The use of diverse programming languages.
- The use of diverse compilers, development tools, and test methods.
- The use of diverse implementation techniques.

8.3. Assumptions

- The failures in the diverse versions are statistically independent.
- All the diverse versions have equal probability of failure $\Rightarrow R_i = R = 1-f$.
- A simple majority voting technique is used in the patterns with voter.
- The voter is carefully designed and can be considered as fault-free.

8.4. Voting Techniques

For the patterns that use a voting component, there are many voting techniques that can be used to implement this voting component such as:

- *Majority Voting*: It is the simplest and most common used method that is used to find the output, where at least $\lceil (N+1)/2 \rceil$ variant results agree.
- *Plurality Voting (PV)*: It is a simple voter, that implements M-out-of-N voting, where M is less than a strict majority.
- *Consensus Voting (CV)* [72]: This voting method is used for multi-version software with small output space. In this method, the result of the largest agreement number is chosen as correct output.
- *Maximum Likelihood Voting (MLV)* [65]: The voter uses the reliability of each version to make a more accurate estimation of the most likely correct result.
- *Adaptive Voting* [49]: It introduces an individual weighting factor to each version which is later included in the voting procedure. These weighting factors are dynamically changeable to model and manage different quality levels of versions.

8.5. N-Version Programming Pattern (NVP)

Other Names:
Master-Slave Pattern [22].

Type:
Software Pattern.

Abstract:
N-Version Programming [13, 11, 25] is a very well known fault-tolerant software method based on software diversity and fault masking. It is defined as the independent generation of $N \geq 2$ functionally equivalent software modules called "versions" from the same initial specification [13]. This pattern includes N programs that are running in parallel to perform the same task on the same input to produce N outputs. A voter is used in this pattern to produce the correct output; it accepts the N results as inputs, and uses these results to determine the correct output according to a specific voting scheme.

Context:
Developing a fault-tolerant software for a highly safety-critical system in a situation where:

- Highly reliable software is needed.
- The high cost for developing multiple implementations can be covered.
- There are available independent N teams to develop the different versions.
- There is a possibility to use N redundant hardware units to execute these versions in parallel.

Problem:
How to overcome the software faults, which may remain after the software development, in order to improve the software reliability and safety.

Pattern Structure:
The N-Version Programming Pattern is a well known fault-tolerant approach that is used for very critical applications due to the high development cost [97]. The idea of this pattern is based on the concept of independent generation of functionally equivalent N versions from the same initial specification. The outputs of these versions are sent to the voter which executes a voting strategy

to determine the best correct output. The goal of N-Version Programming Pattern is to minimize the probability of concurrent faults in these versions.

The structure of this pattern is shown in Fig. 8.1, where the function of each unit is as follows:

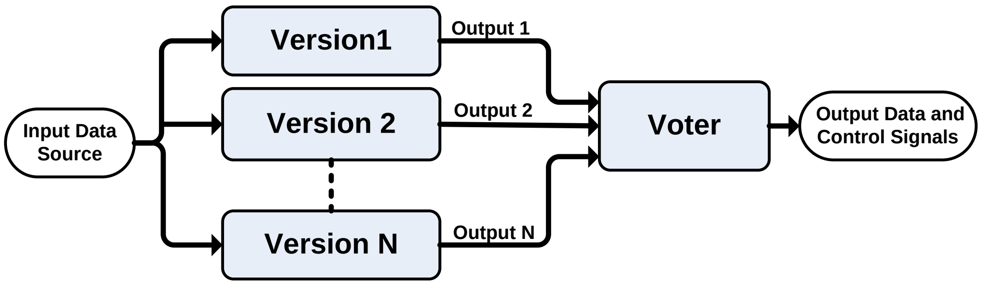


Figure 8.1.: N-Version Programming Pattern

- *Input Data Source*: (see the general pattern in Section 6.7).
- *Output Data and Control Signals*: (see the general pattern in Section 6.7).
- *Version* $1, 2..N$: These versions represent a result of independent implementations of the required software by independent teams of programmers, based on the same initial specification. So, these versions are functionally equivalent and performing the same action on the input data to produce the final result. Usually, these diverse versions are executed in parallel on different hardware devices to generate N outputs which are later processed by the voter to determine the best or correct output data based on a voting strategy. It is also possible to execute these versions sequentially on the same hardware, but this case will increase the time of execution by at least N times, which make this choice less attractive.
- *Voter*: The voter accepts the N outcomes from the diverse versions as inputs, and uses them to determine the final output and control signals based on a specific voting technique such as majority voting which is the most commonly used technique. The majority voting technique is a variant of *M-out-of-N* technique where $M = \lceil \frac{N+1}{2} \rceil$; It produces an output that represents an agreement between at least M versions.

Implication:
This section shows the implication of this pattern using a majority voting relative to the basic system.

- ***Reliability***:
  Under the assumption that the voter is fault-free, the NVP pattern will continue to give a correct result as long as at least M versions have no fault.

$$R_{new} = \sum_{i=M}^{N} \binom{N}{i} (R)^i (1-R)^{N-i} \tag{8.1}$$

  $\Rightarrow$ The percentage relative improvement in software reliability is

$$\begin{aligned} RRI &= \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% \\ RRI &= \frac{\left(\sum_{i=M}^{N} \binom{N}{i} (R)^i (1-R)^{N-i}\right) - R}{1 - R} \times 100\% \end{aligned} \tag{8.2}$$

- ***Safety***:
  The NVP Pattern includes two design techniques: *diverse programming* and *fault detection with voting.* According to the software requirements in the standard IEC 61508-3 [46], the recommendations for these techniques are shown in Tab. 8.2.

Table 8.2.: Recommendations for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Diverse programming	R	R	R	HR
Fault detection with voting	–	R	HR	HR

According to the last table, the average recommendations of this pattern for the different safety integrity levels are shown in Tab. 8.3.

Table 8.3.: Recommendations of NVP Pattern for safety integrity levels.

Pattern	**SIL1**	**SIL2**	**SIL3**	**SIL4**
N-Version Programming Pattern	WR	R	MR	HR

The safety in this pattern depends on whether or not the failed versions produce the same erroneous output which may form a majority in the voting process. In the case of wide output range, it is unlikely that the failed versions will produce similar erroneous results which minimize the probability to find a majority among the erroneous outputs. In this case, it is possible to modify the pattern to force the system to enter its fail-safe state when there is no majority between the outputs. On the other hand, the worst case occurs when M versions produce similar erroneous results as in the logical output. In this case, the erroneous results may produce a majority in the voting process that leads to unsafe system failure.

To compute RSI:

$$P_{UF(old)} = 1 - R = f \tag{8.3}$$

The NVP Pattern will contain unsafe failure only when at least M modules give the same faulty result. Therefore, the probability of unsafe failure in this pattern is

$$P_{UF(new)} = \sum_{i=M}^{N} \binom{N}{i} (1-R)^i (R)^{N-i} \tag{8.4}$$

$\Rightarrow$ The percentage relative safety improvement is

$$\begin{aligned}
RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\
RSI &= \left(1 - \frac{P_{UF(new)}}{P_{UF(old)}}\right) \times 100\% \\
RSI &= \left(1 - \frac{\sum_{i=M}^{N} \binom{N}{i} (1-R)^i (R)^{N-i}}{1-R}\right) \times 100\% \\
RSI &= \left(1 - \frac{\sum_{i=M}^{N} \binom{N}{i} (f)^i (1-f)^{N-i}}{f}\right) \times 100\% \qquad (8.5)
\end{aligned}$$

- ***Cost***:
  This pattern is one of the very expensive patterns. The cost can be divided into two parts.
  - *Recurring Cost*:It includes the cost of N different hardware units to be used for the parallel execution of the N-version software. So, the recurring cost will be ($N \times 100\%$) comparing to the basic system with single-version software.
  - *Development Cost*: The development of independent and functionally equivalent N-Version software can be estimated as follows:
    * The N versions have the same specification, and only one set of specification has to be developed [57].
    * The cost for developing N versions prior to the verification and the validation phase is N times as the cost for developing a single version [11].

  * The different versions can be used to validate each other, which will decrease the cost for verification and validation tools [57].
  * The management of an N-version project imposes overhead not found in traditional software development [57].

  Exact information about the cost of creating N versions from the same specification is limited. The estimated practical cost of development of multi-version software shows that the cost increases sublinearly with the number of components [33]. And each additional version costs about $75 - 80\%$ of a single version [68].

- ***Modifiability***:
  There are three possible modifications
  1. *Modification of a single version*: It is possible to modify a single version either to remove a newly discovered fault, or to improve a poorly programmed function [12]. In this case, the initial specification remains without any modification, and the modification of this version is similar to the modification of single-version software following a standard fault removal procedure.
  2. *Modification of all member versions*: The reason for this modification is either to add a new functionality or to improve the overall performance of the N-version software [12]. In this case, the initial specification should be modified, and all the N versions should be modified and tested independently by independent teams to perform the new changes to the affected modules. In general, the modification of N-version software is remarkably more difficult than the modification of single-version software.
  3. *Modification of the voter*: The separation of the voter from the N versions allows easy modifications or changes of the voting technique.

- ***Impact on Execution Time***:
  There are two methods to execute N-version software, either using parallel execution which is the most common one, or the sequential execution. In the case of parallel execution on N independent hardware units, the different versions might need different execution time since they are implemented by different teams of programmers. Consequently, the voter needs to wait for all outputs to start the voting algorithm. So, the total time of execution is determined by the slowest version in addition to the relatively small time to execute the voting algorithm. In general, if we neglect the voter execution time, then the execution time of the N-version software is slightly equal to single-version software.

In the case of sequential execution, the execution time will increase by N times. This disadvantage makes the sequential execution less attractive, especially for time-critical applications.

Implementation:

- The success of the NVP Pattern depends on the independent development of the required N versions and the level of diversity in these versions to avoid the common failures. In order to increase the level of diversity and the independence of the designed versions, the implementation issues presented in Section 8.2 should be taken into consideration as long as possible during the development process.
- There are several voting techniques that can be used in this pattern to implement the voter component (see Section 8.4). The selection of the suitable technique depends on the data type, deviation in the outputs of the versions, type of agreement [63], output space cardinality size, functionality of the voter [75], reliability of different versions, and many other factors.

Consequences and Side Effects:
The main drawbacks of the NVP Pattern are the complexity of developing independent N-versions in addition to the high development cost and the high dependency on the initial specification which may propagates dependent faults to all versions. The problem of dependent faults in all versions is critical in this pattern for the system safety and reliability.

Related Patterns:
The most related pattern is the Recovery Block Pattern which uses diverse software versions with acceptance tests instead of using a voting algorithm. On the other hand, the N-version programming is used to improve the software reliability, and it uses multiple hardware units. Thus, it is possible to combine this pattern with the Heterogeneous Design Pattern for the design of these diverse hardware units to deal with the systematic hardware faults. This combination will improve the reliability and safety of the hardware as well as the software.

8.6. Recovery Block Pattern (RB)

Other Names:
–

Type:
Software Pattern.

Abstract:
Recovery Block [68, 82, 84, 97] is a very well known fault-tolerant software method. It is based on fault detection with acceptance tests and backward error recovery to avoid system failures. As in N-Version Programming, the Recovery Block Pattern includes N diverse, independent, and functionally equivalent software modules called "versions" from the same initial specification. These modules are classified into primary and $N-1$ secondary versions, where the execution of any version is followed by an acceptance test. The primary version (Block) is firstly executed and followed by its acceptance test. Failure of the test will result in execution of a secondary alternate version (Recovery Block), which is also followed by an acceptance test. The last step is repeated until either one alternate passes its acceptance test or all alternates are executed without passing the test and an overall system failure is reported.

Context:
Developing a fault-tolerant software for a highly safety-critical system in a situation where:

- Highly reliable and safety-critical software is needed.
- There is a possibility to construct an acceptance test to ensure the correct operation of the software and to detect the possible erroneous output.
- There are available independent N teams to develop the different versions.
- The high cost for developing multiple implementations can be covered.

Problem:
How to overcome the software faults, which may remain after the software development, in order to improve the software reliability and safety.

Pattern Structure:
Similar to the NVP, Recovery Block Pattern is a well known fault-tolerant approach that is used for moderately and highly critical applications due to the

high development cost. The difference is that in the Recovery Block Pattern, only a single version is executed at any time. It can be used for complete software or for a critical part of the software. The first idea of this pattern was introduced in the 1970s [82, 84]. It is based on the concept of independent generation of functionally equivalent versions from the same initial specification and the use of fault detection [80].

After the execution of the primary version, the acceptance test (AT) is executed to check if the outcome is reasonable and to detect any possible erroneous result. If the acceptance test is passed, then the outcome is considered as true. Otherwise, if a fault is detected by the acceptance test, the system state should be restored to its original state and an alternate version will be invoked to repeat the same computations. This process is repeated until either one of the alternate secondary versions passes the acceptance test and gives the result, or no more secondary versions are available. In the last case, an overall system failure is reported to execute the available safety action such as switching the system into its fail-safe sate, or to shutdown the system.

The structure of this pattern is shown in Fig. 8.2, where the function of each unit is as follows:

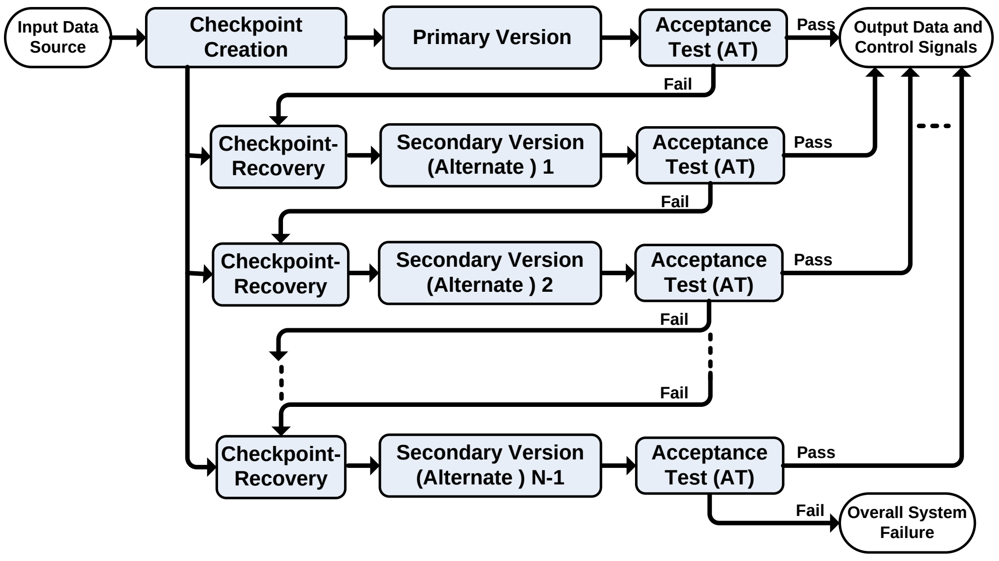


Figure 8.2.: Recovery Block Pattern

- *Input Data Source*: (see the general pattern in Section 6.7).
- *Output Data and Control Signals*: (see the general pattern in Section 6.7).

- *Checkpoint Creation*: In order to provide each alternate with the same data and the same initial conditions, this component is used to create a checkpoint through storing the input data and the system state before executing the primary version. There are many methods and techniques that have been proposed to store the system state such as: recovery cache [82, 84], recoverable procedure [82], and stacked recovery cache [3]. These methods differ in the way of storing the data and the size of the data to be stored, and they include a variation from storing a snapshot of the state of all variables within the system to storing only those values that are about to be changed [97].

- *Checkpoint-Recovery*: This component is used to rollback the system to its initial state when the primary version or the alternate versions fail to pass the acceptance test. This component is invoked before entering any alternate version, where it takes the information about the stored checkpoint to switch the system environment to the same point at which the primary version started computation.

- *Primary and Secondary (Alternate) Versions*: These functionally equivalent versions represent a result of independent implementations of the required software by independent teams of programmers, based on the same initial specification. The primary alternate is the one which is intended to be used normally to perform the desired operation [82]. If a fault is detected in the primary version, then the alternate secondary versions are executed in turn, until one alternate passes its acceptance test, or no more secondary alternate versions are available.

  It is also possible to execute the alternate versions in parallel as in the *Distributed Execution of the Recovery Blocks scheme* [52, 53]. In this scheme, the different versions are distributed on N different hardware units where each unit executes a single version followed by its acceptance test, and the output is normally taken from the primary module. If the primary version fails the acceptance test, the output of the alternate version that passes the acceptance test can be used, and the roles of primary and secondary copies are reversed. Since the alternate versions are executed in parallel, the system does not need to store a checkpoint and to roll back when there is a failure in one version.

- *Acceptance Test (AT)*: It is the part of the software which is executed on the exit from the primary and the alternate versions to confirm that the result is reasonable and to verify the correctness of the calculations based on the software specification. The acceptance test returns true or false, and it may have several components and may include checks for runtime errors [97] and mechanisms for implicit error detection. The success of the recovery block depends on the performance and the quality of the acceptance test. Thus, it should be carefully design and it should be simple, effective, and highly reliable.

Implication:
This section shows the implication of this pattern relative to the basic system.

- ***Reliability***:
  For the success of the recovery block, it must pass the acceptance test with correct outcome at some stage $i, 1 \leq i \leq N$, which means that the test fails the stages from 1 to $i-1$ and at stage i the outcome is correct and it passes the acceptance test.

  Probability{Success at stage i} $= P\{T\}^{i-1} P\{\overline{E} \cap \overline{T}\}$

  R_{new} =Probability that the RB is successful at some stage $i, 1 \leq i \leq N$.

  $$R_{new} = \sum_{i=1}^{N} P\{T\}^{i-1} P\{\overline{E} \cap \overline{T}\} \tag{8.6}$$

  $$\begin{aligned} P\{T\} &= P\{E\}P\{T|E\} + P\{\overline{E}\}P\{T|\overline{E}\} \\ P\{T\} &= fP_{TN} + (1-f)P_{FN} \end{aligned} \tag{8.7}$$

  $$P\{\overline{E} \cap T\} = P\{\overline{E}\}P\{T|\overline{E}\} = (1-f)P_{FN} \tag{8.8}$$

  $$\begin{aligned} P\{\overline{E} \cap \overline{T}\} &= P\{\overline{E}\} - P\{\overline{E} \cap T\} \\ P\{\overline{E} \cap \overline{T}\} &= (1-f) - (1-f)P_{FN} \\ P\{\overline{E} \cap \overline{T}\} &= (1-f)P_{TP} \end{aligned} \tag{8.9}$$

  Under the assumption of $(f = 1 - R)$ and by substituting Equation (8.9) and Equation (8.7) in Equation (8.6) we obtain

  $$R_{new} = \sum_{i=1}^{N} \left((1-R)P_{TN} + RP_{FN}\right)^{i-1} (RP_{TP}) \tag{8.10}$$

  Equation (8.10) shows that the success of the recovery block depends on (P_{TN}) and the (P_{TP}) which are determined by the quality of the acceptance test. Thus, the carefully designed acceptance test is a key factor for the success of this scheme.

$\Rightarrow$ The percentage relative improvement in software reliability is

$$RRI = \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\%$$

$$RRI = \frac{\left(\sum_{i=1}^{N} ((1-R)P_{TN} + RP_{FN})^{i-1} (RP_{TP})\right) - R}{1 - R} \times 100\% \quad (8.11)$$

- ***Safety***:
  The RB Pattern includes three design techniques: *diverse programming*, *fault detection and diagnosis (AT)*, and *recovery block*. According to the software requirements in the standard IEC 61508-3 [46], the recommendations for these techniques are shown in Tab. 8.4.

Table 8.4.: Recommendations for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Diverse programming	R	R	R	HR
Fault detection and diagnosis	–	R	HR	HR
Recovery block	R	R	R	R

According to the last table, the average recommendations of this pattern for the different safety integrity levels are shown in Tab. 8.5.

Table 8.5.: Recommendations of RB Pattern for safety integrity levels.

Pattern	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Recovery Block Pattern	R	R	R	HR

In the case of unsafe failure in the recovery block pattern, it passes the acceptance test with erroneous outcome at some stage $i, 1 \leq i \leq N$, which means that the test fails the stages from 1 to $i-1$ and at stage i the outcome of the version i is erroneous and it passes the acceptance test (false positive case).

Probability{a false positive case at stage i} $= P\{T\}^{i-1} P\{E \cap \overline{T}\}$

$P_{UF(new)}$ =Probability that the *RB* pattern fails unsafely at some stage

$i, 1 \leq i \leq N$.

$$P_{UF(new)} = \sum_{i=1}^{N} P\{T\}^{i-1} P\{E \cap \overline{T}\} \tag{8.12}$$

$$P\{E \cap T\} = P\{E\}P\{T|E\} = fP_{TN} \tag{8.13}$$

$$\begin{aligned} P\{E \cap \overline{T}\} &= P\{E\} - P\{E \cap T\} \\ P\{E \cap \overline{T}\} &= f - fP_{TN} = fP_{FP} \end{aligned} \tag{8.14}$$

By substituting Equation (8.14) and (8.7) in Equation (8.12) we obtain

$$P_{UF(new)} = \sum_{i=1}^{N} (fP_{TN} + (1-f)P_{FN})^{i-1} (fP_{FP}) \tag{8.15}$$

$\Rightarrow$ The percentage relative safety improvement is

$$\begin{aligned} RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\ RSI &= \left(1 - \frac{P_{UF(new)}}{P_{UF(old)}}\right) \times 100\% \\ RSI &= \left(1 - \frac{\sum_{i=1}^{N} (fP_{TN} + (1-f)P_{FN})^{i-1} (fP_{FP})}{f}\right) \times 100\% \end{aligned} \tag{8.16}$$

- ***Cost***:
  The cost can be divided into two parts.
  - *Recurring Cost*:The normal recovery block runs the independent versions serially on a single hardware unit. Thus, there is no extra hardware requirement for this pattern except the memory location to store the recovery checkpoint.

    For distributed recovery block scheme, the independent versions are executed in parallel on N different hardware units. So, the recurring cost will be ($N \times 100\%$) comparing to the basic system with single-version software.
  - *Development Cost*: The development cost of the recovery block approach is approximately equivalent to the development cost of NVP Pattern since it includes the development of independent and functionally equivalent N versions. The estimated practical cost, which can be conducted in the same way as for NVP, shows that the cost

increases sublinearly with the number of components [33]. And each additional version costs about $75-80\%$ of a single version [68]. Moreover, this pattern includes extra cost for developing an effective acceptance test, which has great impact on the success of this pattern.

- ***Modifiability***:
  There are three possible modifications

  1. *Modification of a single version*: It is possible to modify a single version either to remove a newly discovered fault, or to improve a poorly programmed function [12]. In this case, the initial specification remains without any modification, and the modification of this version is similar to the modification of single-version software following a standard fault removal procedure.
  2. *Modification of all member versions*: The reason for this modification is either to add a new functionality or to improve the overall performance of the N-version software [12]. In this case, the initial specification should be modified, and all the N versions should be modified and tested independently by independent teams to perform the new changes to the affected modules. In general, the modification of N-version software is remarkably more difficult than the modification of single-version software.
  3. *Modification of the acceptance test*: The acceptance test can be considered as independent module. Thus, it can be easily modified without any influence on the different versions.

- ***Impact on Execution Time***:
  The total execution time of a single version followed by its AT is

$$T = t_{SV} + t_{AT} \tag{8.17}$$

Where:
$-$ t_{SV} : the execution time of a single version.
$-$ t_{AT}: the execution time of the acceptance test.

Most of the time, only the primary alternate of the recovery block is executed, which keeps the run-time overhead of the recovery block to a minimum value which represents the best execution time (T) [85]. In the worst case, all the alternates are executed without getting a reasonable result. Under the assumption that the different versions have equal execution times, the worst case execution time will be $N \times T$.

Consequently, the worst case execution time of the RB Pattern can be considered as the main disadvantage which makes it less suitable for time-critical applications.

Implementation:
The success of the RB pattern depends on two factors [57]:

1. The quality of the acceptance test. Therefore, the acceptance test should be carefully designed to detected most of the possible software faults, which will improve the reliability of the recovery blocks.
2. The independent development of the required N versions and the level of diversity in these versions to avoid the common failures. In order to increase the level of diversity and the independence of the developed versions, the implementation issues presented in Section 8.2 should be taken into consideration as long as possible during the development process.

Consequences and Side Effects:
The main drawbacks of the RB are the high dependency on the quality of the acceptance test and the time influence of the sequential execution which may include a situation with interrupted service during recovery. Also, It has common drawbacks with the NVP Pattern, such as the complexity of developing independent N-versions, the high development cost, and the high dependency on the initial specification which may propagate software bugs to all versions.

Related Patterns:
The Recovery Block Pattern is normally combined with the Watchdog Pattern to test the software versions for timeout condition and to provide program sequence monitoring. The distributed Recovery Block runs the alternate versions in parallel on different hardware units. Thus, it is possible to combine the distributed Recovery Block with the Heterogeneous Design Pattern for the design of these diverse hardware units to deal with the systematic hardware faults.

8.7. Acceptance Voting Pattern (AV)

Other Names:
–

Type:
Software Pattern.

Abstract:
Acceptance Voting Pattern [10] is a hybrid pattern that incorporates the NVP Pattern with the acceptance test used by the RB Pattern. Similar to the NVP, this pattern is based on the independent generation of $N \geq 2$ functionally equivalent software modules called “versions” from the same initial specification [13]. This pattern includes N programs that are running in parallel to perform the same task on the same input to produce N outputs. The output of each version is presented to an acceptance test to check it for correctness. The outputs that pass the acceptance test are used as inputs to a dynamic voter, which is executed to produce the correct output according to a voting scheme.

Context:
Developing a fault-tolerant software for a highly safety-critical system in a situation where:

- Highly reliable software is needed.
- There is a possibility to construct an acceptance test to ensure the correct operation of the software and to detect the possible erroneous output.
- The high cost for developing multiple implementations can be covered.
- There are available independent N teams to develop the different versions.
- There is a possibility to use N redundant hardware units to execute these versions in parallel.

Problem:
How to overcome the software faults, which may remain after the software development, in order to improve the software reliability and safety.

Pattern Structure:
The Acceptance Voting Pattern represents a combination of the fault detection scheme provided by the acceptance test and the fault masking scheme provided

by NVP with voting. It includes N independent and functionally equivalent versions that are typically executed in parallel to perform the required task. The output of each version is tested for correctness using an acceptance test. Those results that pass the acceptance test are then used by the voting algorithm to generate the final result.

The structure of this pattern is shown in Fig. 8.3, where the function of each unit is as follows:

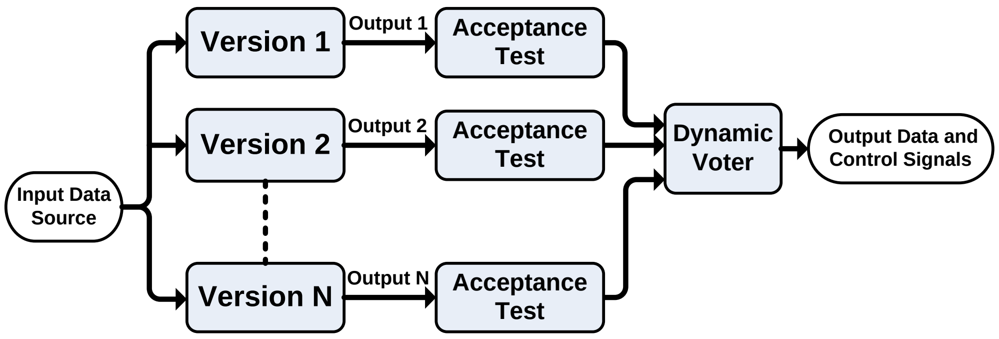


Figure 8.3.: Acceptance Voting Pattern

- *Input Data Source*: (see the general pattern in Section 6.7).
- *Output Data and Control Signals*: (see the general pattern in Section 6.7).
- *Version* $1, 2..N$: These versions represent a result of independent implementations of the required software by independent teams of programmers, based on the same initial specification. So, these versions are functionally equivalent and performing the same action on the input data to produce the final result. These diverse versions are executed in parallel on different hardware devices to generate N outputs which are later presented to an acceptance test to check them for correctness. Those results that pass the test are processed by the dynamic voter to determine the final output.
- *Acceptance Test (AT)*: It is the part of the software which is executed on the outcome of each version to confirm that the result is reasonable and fulfills the defined requirements given in the software specification. The acceptance test returns true or false, and it may have several components and may include checks for runtime errors [97] and mechanisms for implicit error detection. Various implementations of the acceptance test are possible ranging from simple reasonableness checks to complex high-coverage validators [76].

- *Dynamic Voter*: The voter accepts the outputs that pass the acceptance test, and uses theme as inputs to the voting algorithm in order to determine the final output and control signals. The voter in this pattern should be dynamic due to the variable number of inputs that ranges from 0 to N. It may include the following different actions according to the number of outputs that pass the acceptance test:
  - When no output passes the acceptance test, it reports an overall system failure.
  - In the case of one output, it just forwards this output.
  - When two outputs pass the acceptance test, the signal is only forwarded if both are equal (or the difference is within a defined tolerance). For inequality, the action depends on the required level of safety and reliability, either an output is selected according to a predefined order or an exception is raised to indicate a failure.
  - When the number of outputs that pass the acceptance test is more than two, a voting technique is executed to generate the final result.

Implication:
This section shows the implication of this pattern using a majority voting relative to the basic system.

- ***Reliability***:
  If the number of outputs that pass the acceptance test and participate in the voting is n , then these outputs can be grouped into two groups:
  - *Correct (True Positive) Outputs* with probability $= RP_{TP}$.
  - *Incorrect (False Positive) Outputs* with probability $= (1 - R)P_{FP}$.

  The probability that an output passes the test is

$$P\{\overline{T}\} = RP_{TP} + (1 - R)P_{FP} \tag{8.18}$$

  The probability that an output does not pass the test is

$$P\{T\} = RP_{FN} + (1 - R)P_{TN} \tag{8.19}$$

  The probability that the voter gives a correct output, given that n outputs passed the test, is equal to

$$\sum_{i=M}^{n} \binom{n}{i} (RP_{TP})^i \left((1 - R)\, P_{FP}\right)^{n-i} \tag{8.20}$$

The probability that n outputs from the total number of outputs (N) pass the acceptance test and give a correct result in the voting is equal to

$$\binom{N}{n}\left(\sum_{i=M}^{n}\binom{n}{i}(RP_{TP})^{i}\left((1-R)\,P_{FP}\right)^{n-i}\right)\left(RP_{FN}+(1-R)P_{TN}\right)^{N-n} \tag{8.21}$$

The number of versions n that pass the acceptance to participate in the voting can be $1, 2..N$

$$R_{new} = \sum_{n=1}^{N}\left[\binom{N}{n}\left(\sum_{i=M}^{n}\binom{n}{i}(RP_{TP})^{i}\left((1-R)\,P_{FP}\right)^{n-i}\right)\left(RP_{FN}+(1-R)P_{TN}\right)^{N-n}\right] \tag{8.22}$$

$\Rightarrow$ The percentage relative improvement in software reliability is

$$\begin{aligned} RRI &= \frac{R_{new}-R_{old}}{1-R_{old}}\times 100\% \\ RRI &= \frac{R_{new}-R}{1-R}\times 100\% \end{aligned} \tag{8.23}$$

The substitution of Equation (8.22) into Equation (8.23) shows that the reliability improvement in this pattern depends on the reliability and number of versions, and on the effectiveness of the used acceptance test.

- ***Safety***:
The AV Pattern includes two design techniques: *diverse programming* and *fault detection with voting and acceptance test.* According to the software requirements in the standard IEC 61508-3 [46], the recommendations for these techniques are shown in Tab. 8.6.

Table 8.6.: Recommendations for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Diverse programming	R	R	R	HR
Fault detection and diagnosis (Voting and Acceptance Test)	–	R	HR	HR

According to the last table, the average recommendations of this pattern for the different safety integrity levels are shown in Tab. 8.7.

Table 8.7.: Recommendations of AV Pattern for safety integrity levels.

Pattern	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Acceptance Voting Pattern	WR	R	MR	HR

The Probability that the voter gives an erroneous output, given that n outputs passed the test, is equal to

$$\sum_{i=0}^{M-1} \binom{n}{i} \left((1-f)\,P_{TP}\right)^{i} \left(fP_{FP}\right)^{n-i} \tag{8.24}$$

The probability that n outputs from the total number of outputs (N) pass the acceptance test and give an erroneous result in the voting is equal to

$$\binom{N}{n}\left(\sum_{i=0}^{M-1}\binom{n}{i}\left((1-f)\,P_{TP}\right)^{i}\left(fP_{FP}\right)^{n-i}\right)\left((1-f)\,P_{FN}+fP_{TN}\right)^{N-n} \tag{8.25}$$

The number of versions n that pass the acceptance to participate in the voting can be $1,2..N$. Therefore, the probability that the AV fails unsafely is equal to the probability that the voter finds a majority between the erroneous results

$$\begin{aligned} P_{UF(new)} = & \sum_{n=1}^{N}\left[\binom{N}{n}\left(\sum_{i=0}^{M-1}\binom{n}{i}\left((1-f)\,P_{TP}\right)^{i}\left(fP_{FP}\right)^{n-i}\right)\right. \\ & \left.\left((1-f)\,P_{FN}+fP_{TN}\right)^{N-n}\right] \end{aligned} \tag{8.26}$$

$\Rightarrow$ The percentage relative safety improvement is

$$\begin{aligned} RSI &= \frac{P_{UF(new)}-P_{UF(old)}}{0-P_{UF(old)}}\times 100\% \\ RSI &= \left(1-\frac{P_{UF(new)}}{f}\right)\times 100\% \end{aligned} \tag{8.27}$$

- ***Cost***:
  This pattern is resulting in high costs which can be divided into two parts.
  - <u>*Recurring Cost*</u>: It includes the cost of N different hardware units to be used for the parallel execution of the N-version software. So, the recurring cost will be ($N \times 100\%$) comparing to the basic system with single-version software.

  - *Development Cost*: The development cost of the AV Pattern is approximately equivalent to the development cost of the NVP Pattern since it includes the development of independent and functionally equivalent N versions. The estimated practical cost, which can be conducted in the same way as for NVP, shows that the cost increases sublinearly with the number of components [33]. And each additional version costs about $75 - 80\%$ of a single version [68]. Moreover, this pattern includes extra cost for developing an effective acceptance test.

- ***Modifiability***:
  There are four possible modifications
  1. *Modification of a single version*: It is possible to modify a single version either to remove a newly discovered fault, or to improve a poorly programmed function [12]. In this case, the initial specification remains without any modification, and the modification of this version is similar to the modification of single-version software following a standard fault removal procedure.
  2. *Modification of all member versions*: The reason for this modification is either to add a new functionality or to improve the overall performance of the N-version software [12]. In this case, the initial specification should be modified, and all the N versions should be modified and tested independently by independent teams to perform the new changes to the affected modules. In general, the modification of N-version software is remarkably more difficult than the modification of single-version software.
  3. *Modification of the acceptance test*: The acceptance test can be considered as independent module. Thus, it can be easily modified without any influence on the different versions or the voter.
  4. *Modification of the dynamic voter*: The separation of the voter from the N versions and the acceptance test allows easy modifications or changes of the voting technique.

- ***Impact on Execution Time***:
  The diverse software versions in this pattern are executed in parallel, ideally on N independent hardware devices. As the execution times of these software versions might differ since they are implemented independently by different teams of programmers, the voter has to wait for the outputs of all versions to be checked by the acceptance test before applying the voting algorithm. Thus, the total execution time is determined by the

slowest version in addition to the typically relatively small time to execute the acceptance test and the voting algorithm. In general, if we can neglect the execution time of the acceptance test and the voter, then the execution time of this pattern is slightly equal to single-version software.

Implementation:
The success of the AV Pattern depends on three factors:

1. The quality of the acceptance. Thus, it should be carefully designed to detected most of the possible software faults.
2. The independent development of the required *N* versions and the level of diversity in these versions to avoid the common failures. In order to increase the level of diversity and the independence of the developed versions, the implementation issues presented in Section 8.2 should be taken into consideration as long as possible during the development process.
3. The use of a suitable voting technique. There are several voting techniques that can be used in this pattern to implement the dynamic voter component (see Section 8.4). The selection of the suitable technique depends on the data type, deviation in the outputs of the versions, type of agreement [63], output space cardinality size, functionality of the voter [75], reliability of different versions, and many other factors.

Consequences and Side Effects:
Similar to the original N-version programming approach, the drawbacks of the AV Pattern are seen in the effort of developing *N* diverse software versions in addition to the high dependency on the initial specification which may propagate faults to all versions. With respect to safety, the problem of dependent faults in all versions is less critical in this pattern since the acceptance test represents an additional measure to detect these faults.

Related Patterns:
As the AV Pattern is executed on different hardware devices, it is possible to combine this pattern with the Heterogeneous Design Pattern for the design of diverse hardware units that will be able to cope with systematic hardware faults.

8.8. N-Self Checking Programming Pattern (NSCP)

Other Names:
–

Type:
Software Pattern.

Abstract:
The N-Self Checking Programming is one of the most costly fault-tolerant software methods. It is based on software design diversity and self-checking provided by adding redundancy to a program so that it can check its own dynamic behavior during execution [59]. This pattern includes an independent generation of $N \geq 4$ functionally equivalent software modules called "versions" from the same initial specification. These versions are arranged into groups called *components*, in which each component consists of two versions and a comparison algorithm to compare the results of two versions for correctness. During the execution, one component is working as an active component to deliver the required service, while the other components are hot spares. In order to provide a fault tolerance for a single fault, at least 4 versions must be executed on 4 hardware units, which represents the highest cost comparing to the other methods.

Context:
Developing a fault-tolerant software for a highly safety-critical system in a situation where:

- Highly reliable software is needed.
- The high cost for developing multiple implementations can be covered.
- There are available independent N teams to develop the different versions.
- There is a possibility to use N redundant hardware units to execute these versions in parallel.

Problem:
How to overcome the software faults, which may remain after the software development, in order to improve the software reliability and safety.

Pattern Structure:
The NSCP Pattern is a well known fault-tolerant approach that is based on

the concept of software design diversity and error detection by self-checking programming. It involves a parallel execution of at least two self-checking components arranged in hot standby redundancy, where each component includes two independent and functionally equivalent versions that run in parallel and are self checked using a comparison algorithm. When the running component fails due to different results from its versions, a spare component is invoked to start delivering the required functionality. In safety-critical applications, when all spare components fail, either an overall system failure is reported or a signal is generated to run the available safety function.

The structure of this pattern is shown in Fig. 8.4, where the function of each unit is as follows:

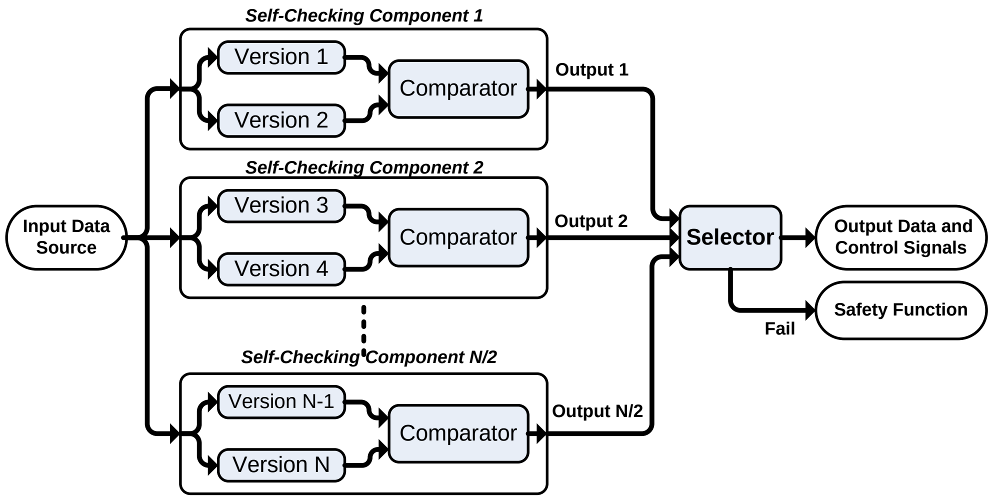


Figure 8.4.: N-Self Checking Programming Pattern

- *Input Data Source*: (see the general pattern in Section 6.7).
- *Output Data and Control Signals*: (see the general pattern in Section 6.7).
- *Self-Checking Component*: It is the main building part in this pattern, where each self-checking component consists of two software versions and a comparator. The software dynamic behavior is checked during the execution through applying a comparison algorithm to the results of the two internal versions, which are executed in parallel to perform the required functionality. The output of each component consists of two parts: the computed output data and a signal to indicate the status of the component. If there is no agreement between the two versions, then this component is discarded and a signal is generated to indicate a fault in this component.

- *Version* $1, 2..N$: These versions represent a result of independent implementations of the required software by independent teams of programmers, based on the same initial specification. So, these versions are functionally equivalent and performing the same action on the input data to produce the final result. Typically, the number of versions N is an even number in order to group theme in pairs to construct $N/2$ self-checking components.
- *Comparator*: This part is used in each self-checking component as a method for fault detection. It applies a comparison algorithm to the results of two versions in order to check theme for correctness. If there is an agreement between the two results, then it generates a valid signal and delivers the result to the selector. If there is no agreement, the result of the component is discarded and a signal is sent to indicate a failure in this component. The same comparison algorithm is normally used in all components. Nevertheless, the design diversity could be also applied to develop a separate comparator for each component.
- *Selector*: It is a simple module used to switch between the self-checking components in hot standby redundancy mode. It receives a status signal and output data from each component. If there is a disagreement signal from the acting component, then it switches to the next spare component. If the selector reaches the last component without getting an agreement, then it issues a signal to start executing the available safety function.
- *Safety Function*: It includes the actions to be conducted in the case of failure in all components. It may include different actions that range from switching the system into its fail-safe sate to a complex safety algorithm that involves a predefined sequence of safety steps.

Implication:
This section shows the implication of this pattern using a majority voting relative to the basic system.

- ***Reliability***:
  We will use the following notations for this pattern:
  - C_j: The self-checking component number j.
  - R_C: The reliability of a self-checking component.
  - E_C: The event that a self-checking component fails.

  For the success of the NSCP, the comparator has to find an agreement at some component $C_j, 1 \leq j \leq N/2$, which means that the components from 1 to $j-1$ failed, and the component j gives a correct result.

Probability {Success in a component} = Probability that the two versions in this component give the same correct result.

$$R_c = P\{\overline{E_C}\} = R^2 \tag{8.28}$$

The probability that the NSCP is successful at some component C_j, $1 \leq j \leq N/2$ is

$$\begin{aligned} R_{new} &= \sum_{j=1}^{N/2} P\{E_C\}^{j-1} P\{\overline{E_C}\} \\ R_{new} &= \sum_{j=1}^{N/2} \left(1-R^2\right)^{j-1} \left(R^2\right) \\ R_{new} &= 1-\left(1-R^2\right)^{N/2} \end{aligned} \tag{8.29}$$

$\Rightarrow$ The percentage relative improvement in software reliability is

$$\begin{aligned} RRI &= \frac{R_{new}-R_{old}}{1-R_{old}} \times 100\% \\ RRI &= \frac{\left(1-\left(1-R^2\right)^{N/2}\right)-R}{1-R} \times 100\% \\ RRI &= \left(1-\frac{\left(1-R^2\right)^{N/2}}{1-R}\right) \times 100\% \end{aligned} \tag{8.30}$$

- ***Safety***:
  The NSCP Pattern includes two design techniques: *diverse programming* and *fault detection and diagnosis with a comparator*. According to the software requirements in the standard IEC 61508-3 [46], the recommendations for these techniques are shown in Tab. 8.8.

Table 8.8.: Recommendations for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Diverse programming	R	R	R	HR
Fault detection and diagnosis with a comparator	–	R	HR	HR

According to the last table, the average recommendations of this pattern for the different safety integrity levels are shown in Tab. 8.9.

Table 8.9.: Recommendations of NSCP Pattern for safety integrity levels.

Pattern	**SIL1**	**SIL2**	**SIL3**	**SIL4**
N-Self Checking Programming Pattern	WR	R	MR	HR

The safety of this pattern depends on the probability that the comparison algorithm finds an agreement on an erroneous output at some component $C_j, 1 \leq j \leq N/2$, which means that the components from 1 to $j-1$ failed without an agreement, and the two versions inside the component j give the same erroneous result. Similar to the NVP Pattern, this issue is highly dependent on the type of output data, and on the probability that two versions give identical erroneous results.

Assume that we have a binary output, which means that the two versions inside each component fail in the same manor which represents the worst case, and the computed value for P_{UF} represents the upper limit.

To compute RSI:

$$P_{UF(old)} = 1 - R = f \tag{8.31}$$

The probability that a component fails without agreement is equal to the probability that one version produces a correct result and one version fails

$$= 2\left(1 - R\right) R = 2f\left(1 - f\right) \tag{8.32}$$

Probability that the two versions give the same erroneous result is

$$= \left(1 - R\right)\left(1 - R\right) = f^2 \tag{8.33}$$

Probability that the NSCP fails unsafely at some component $C_j, 1 \leq j \leq N/2$

$$\begin{aligned} P_{UF(new)} &= \sum_{j=1}^{N/2} \left(2f - 2f^2\right)^{j-1} f^2 \\ P_{UF(new)} &= f^2 \left(\frac{1 - \left(2f - 2f^2\right)^{N/2}}{1 - \left(2f - 2f^2\right)} \right) \end{aligned} \tag{8.34}$$

$\Rightarrow$ The percentage relative safety improvement is

$$\begin{aligned} RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\ RSI &= \left(1 - \frac{P_{UF(new)}}{f}\right) \times 100\% \\ RSI &= \left(1 - f\left(\frac{1 - \left(2f - 2f^2\right)^{N/2}}{1 - (2f - 2f^2)}\right)\right) \times 100\% \end{aligned} \tag{8.35}$$

- ***Cost***:
  The cost can be divided into two parts.
  - *Recurring Cost*:It includes the cost of N different hardware units to be used for the parallel execution of the N-version software. So, the recurring cost will be ($N \times 100\%$) comparing to the basic system with single-version software. Moreover, the recurring cost will also include the cost of the comparator and selector unit.
  - *Development Cost*: The development cost of the NSCP is approximately equivalent to the development cost of the NVP Pattern since it includes the development of independent and functionally equivalent N versions. The estimated practical cost, which can be conducted in the same way as for NVP, shows that the cost increases sublinearly with the number of components [33]. And each additional version costs about $75 - 80\%$ of a single version [68].Moreover, this pattern includes extra cost for developing the comparator and selector unit.

  In general, this pattern is considered as the most expensive among the patterns that share the principle of diversity programming. To provide a fault tolerance for a single fault, it needs 4 versions to run on 4 hardware modules in parallel, which is considered as the highest cost.

- ***Modifiability***:
  There are three possible modifications
  1. *Modification of a single version*: It is possible to modify a single version either to remove a newly discovered fault, or to improve a poorly programmed function [12]. In this case, the initial specification remains without any modification, and the modification of this version is similar to the modification of single-version software following a standard fault removal procedure.

2. *Modification of all member versions*: The reason for this modification is either to add a new functionality or to improve the overall performance of the N-version software [12]. In this case, the initial specification should be modified, and all the N versions should be modified and tested independently by independent teams to perform the new changes to the affected modules. In general, the modification of N-version software is remarkably more difficult than the modification of single-version software.
3. *Modification of the comparator*: The separation of the comparator module from the N versions allows easy modifications or changes of the voting technique.

Besides the previous possible major modifications, there is a possibility to modify the self-checking components to rearrange the different versions. The best two versions can be grouped in the same component to provide a strong one that can be activated most of the time with high reliability.

- ***Impact on Execution Time***:
  The independent versions, which are executed in parallel, might need different execution time since they are implemented by different teams of programmers. As the comparator needs to wait for the results of the two versions to start the comparison algorithm, the total execution time for a single component is determined by the slowest version in this component and the relatively small time to execute the comparison algorithm.

  The spare self-checking components are working in a hot standby mode and do not need any time for booting in the presence of a fault in the acting component. In the worst case, this mode has the least amount of influence on the time of execution since it just includes the small time to switch from the acting component to another spare.

  In general, if we neglect the execution time of the comparator and the time for switching, then the execution time of the NSCP method is slightly equal to single-version software.

Implementation:

- The parallel execution of the components necessitates a consistency of inputs and outputs [59]. Thus, a consistency mechanism should be included in the implementation of this pattern to provide synchronization in inputs and outputs.
- The comparator and selector component should be carefully designed to provide an efficient comparison and fast switching.

- The success of the NSCP Pattern depends on the independent development of the required N versions and the level of diversity in these versions to avoid the common failures. In order to increase the level of diversity and the independence of the designed versions, the implementation issues presented in Section 8.2 should be taken into consideration as long as possible during the development process.
- The implementation of the self-checking component in the original proposed approach [59] includes two methods: the use of two versions and a comparison algorithm as listed above, or the use of a single version followed by an acceptance test. Actually, the second methods is identical to the distributed recovery block approach, in which the different versions are executed in parallel and followed by the execution of the acceptance tests to check the correctness of versions.

Consequences and Side Effects:
The NSCP Pattern has the following drawbacks:

- High dependency on the initial specifications that may propagate faults to all versions.
- The high number of diverse versions and hardware modules used comparing to the other patterns that tolerate the same number of faults.
- The complexity of developing N independent and functionally equivalent versions.

Related Patterns:
The NSCP Pattern is used to improve the software reliability by executing the diverse versions in parallel. Thus, it is possible to combine this pattern with the Heterogeneous Design Pattern for the design of diverse hardware units to deal with the systematic hardware faults.

8.9. Recovery Block with Backup Voting Pattern (RBBV)

Other Names:
–

Type:
Software Pattern.

Abstract:
The Recovery Block with Backup Voting [8] is a hybrid pattern that incorporates the idea of the classical RB with the NVP to improve the reliability of the classical RB in those situations where it is difficult to construct an effective acceptance test. This software pattern includes N diverse, independent, and functionally equivalent software modules called "versions" from the same initial specification. It solves the problem of false negative cases which include a wrong consideration of correct outputs by the acceptance test as erroneous outputs. The primary version is firstly executed and followed by its acceptance test. When the first version fails to pass the acceptance test, a copy of the outcome is stored in a cache memory as a backup and the next alternate version is invoked to perform the required functionality. The last step is repeated until either one alternate passes its acceptance test. If all alternates are executed without passing the acceptance test, the stored values will be used as inputs to a voting method as a last attempt to deliver a valid result.

Context:
Developing a fault-tolerant software for a highly safety-critical system in a situation where:

- Highly reliable and safety-critical software is needed.
- It is difficult to construct an effective acceptance test to ensure the correct operation of the software and to detect the possible faults.
- There are available independent N teams to develop the different versions.
- The high cost for developing multiple implementations can be covered.

Problem:
How to overcome the software faults, which may remain after the software development, in order to improve the software reliability, and how to solve the problem of false negative cases in weak acceptance tests.

Pattern Structure:
The Recovery Block with Backup Voting represents a combination of the fault detection provided by the acceptance test in the RB Pattern and the fault masking provided by the voter in the NVP Pattern. It can be considered as a solution to the problem of false negative cases of the weak acceptance test, which includes a wrong consideration of correct outputs by the acceptance test as erroneous outputs (see Section 5.2 and 5.3).

After the execution of the first version, the acceptance test is executed to check the outcome for reasonableness and to detect any possible erroneous result. If the acceptance test is passed, then the outcome is considered as valid. Otherwise, if a fault is detected by the acceptance test, a copy of the outcome is stored as backup in a cache memory location to be used later, then the system state should be restored to its original state and an alternate version will be invoked to repeat the same computations. This process is repeated until either one of the alternate versions passes the acceptance test and gives the result, or no more versions are available. When the last alternate version fails to pass the acceptance test, there are two possible reasons: either all the versions fail concurrently, or there is a failure in the acceptance test. Unlike the classical RB, a voting technique is executed using stored backup values as a last attempt to deliver a valid result and to distinguish between the two possible reasons.

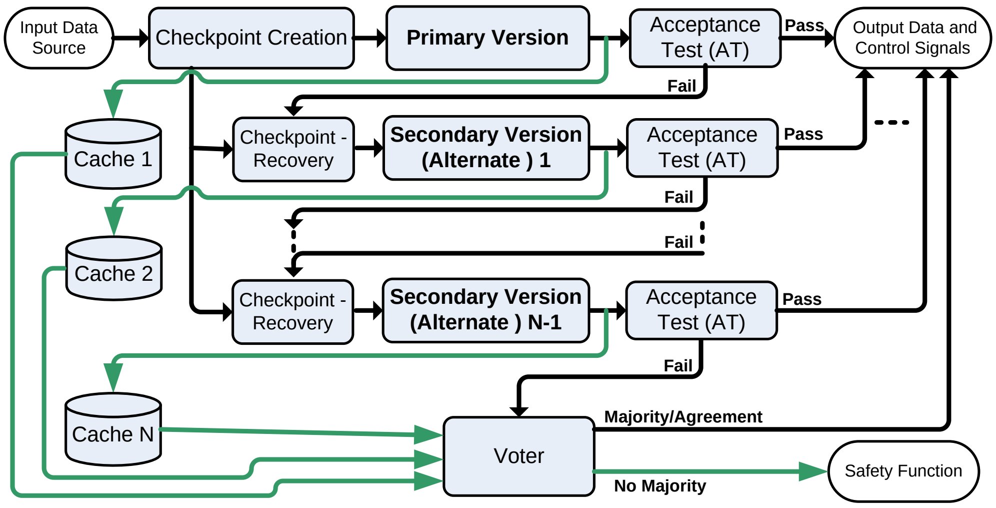


Figure 8.5.: Recovery Block with Backup Voting Pattern

The structure of this pattern is shown in Fig. 8.5, where the function of each unit is as follows:

- *Input Data Source*: (see the general pattern in Section 6.7).
- *Output Data and Control Signals*: (see the general pattern in Section 6.7).
- *Checkpoint Creation*: In order to provide each alternate with the same data and initial conditions, this component is used to create a checkpoint through storing the system state before executing the primary version.
- *Checkpoint-Recovery*: This component is used to rollback the system to its initial state when the primary version or alternate versions fail to pass the acceptance test. This component is invoked before entering any alternate version, where it takes the information about the stored checkpoint to switch the system environment to the same point at which the primary version started computation.
- *Primary and Secondary (Alternate) Versions*: These functionally equivalent versions represent a result of independent implementations of the required software by independent teams of programmers, based on the same initial specification. The primary alternate is the one which is intended to be normally used to perform the desired operation. If a fault is detected in the primary version, then the alternate secondary versions are executed in turn, until one alternate passes its acceptance test, or no more secondary alternate versions are available.

  In this pattern, a copy of the result of each failed version is stored in a cache memory to be later used as input to a voting technique when all the versions fail to pass the acceptance test.
- *Acceptance Test (AT)*: It is the part of the software which is executed on the exit from the alternate versions to confirm the correctness of the calculations based on the software specification. The acceptance test returns true or false, and it may have several components and may include checks for runtime errors [97] and mechanisms for implicit error detection. The acceptance test can be implemented in different variants that range from simple reasonableness checks to complex high-coverage validators, but this pattern is more suitable for the situations that include weak acceptance tests or it is difficult to develop an effective one. Nevertheless, the success of this pattern still depends on the performance and the quality of the acceptance test, and it should be carefully design and it should be simple, effective as much as possible.
- *Cache Memory*: This unit includes memory locations to store the results of the versions that fail the acceptance test. The size of these locations depend on the size of the output data and the number of alternate versions.

- *Voter*: The voter is executed when the last version fails to pass the acceptance test as a last attempt to deliver a valid result using the stored backup values as inputs to the voting. The selection of the voting technique between majority and agreement depends on the required level of accuracy and reliability. If the voter finds a majority between the backup values, it will consider this majority value as a correct output. Otherwise, if there is no majority between the backup values, the voter will consider that the problem is in the alternate versions and not in the acceptance test. In this case, either an overall system failure is reported or a signal is generated to run the available safety function.

- *Safety Function*: It includes the actions to be conducted as a response to an overall failure in the system. It may include different actions that range from switching the system into its fail-safe sate to a complex safety algorithm that involves a predefined sequence of safety steps.

Implication:
This section shows the implication of this pattern relative to the basic system.

- ***Reliability***:
  For the success of the RBBV, it must give a valid result in one of the following two cases:

  1. An alternate version passes the acceptance test with correct outcome at some stage i, $1 \leq i \leq N$, which means that the test fails the stages from 1 to $i-1$ and at stage i the outcome passes the test.
  2. The voter finds a correct majority among the stored values.

  ◦ *In the First Case*:
  Probability{Success at stage i} $= P\{T\}^{i-1}P\{\overline{E} \cap \overline{T}\}$
  The probability that the $RBBV$ is successful at some stage i, $1 \leq i \leq N$

$$= \sum_{i=1}^{N} P\{T\}^{i-1}P\{\overline{E}\cap\overline{T}\} \tag{8.36}$$

$$\begin{aligned} P\{T\} &= P\{E\}P\{T|E\} + P\{\overline{E}\}P\{T|\overline{E}\} \\ P\{T\} &= fP_{TN} + (1-f)P_{FN} \end{aligned} \tag{8.37}$$

$$P\{\overline{E} \cap T\} = P\{\overline{E}\}P\{T|\overline{E}\} = (1-f)P_{FN} \tag{8.38}$$

$$
\begin{aligned}
P\{\overline{E} \cap \overline{T}\} &= P\{\overline{E}\} - P\{\overline{E} \cap T\} \\
P\{\overline{E} \cap \overline{T}\} &= (1-f) - (1-f)P_{FN} \\
P\{\overline{E} \cap \overline{T}\} &= (1-f)P_{TP} \qquad (8.39)
\end{aligned}
$$

By substituting Equation (8.39) and Equation (8.37) in Equation (8.36), the probability that the *RBBV* is successful at some stage i

$$
= \sum_{i=1}^{N} \left(fP_{TN} + (1-f)P_{FN}\right)^{i-1} \left((1-f)P_{TP}\right) \qquad (8.40)
$$

∘ <u>*In the Second Case*</u>:
The probability that a backup value includes a correct result is

$$
\begin{aligned}
P\{correct\} &= P\{\overline{E}\}P\{T|\overline{E}\} \\
P\{correct\} &= (1-f)P_{FN} \qquad (8.41)
\end{aligned}
$$

The probability that a backup value includes an erroneous result is

$$
\begin{aligned}
P\{erroneous\} &= P\{E\}P\{T|E\} \\
P\{erroneous\} &= fP_{TN} \qquad (8.42)
\end{aligned}
$$

The probability to find a correct majority between the backup values is

$$
\begin{aligned}
&= \sum_{i=M}^{N} \binom{N}{i} \left(P\{correct\}\right)^{i} \left(P\{erroneous\}\right)^{N-i} \\
&= \sum_{i=M}^{N} \binom{N}{i} \left((1-f)P_{FN}\right)^{i} \left(fP_{TN}\right)^{N-i} \qquad (8.43)
\end{aligned}
$$

By adding Equation (8.40) and Equation (8.43) we obtain

$$
\begin{aligned}
R_{new} &= \left(\sum_{i=1}^{N} \left(fP_{TN} + (1-f)P_{FN}\right)^{i-1} \left((1-f)P_{TP}\right)\right) \\
&\quad + \left(\sum_{i=M}^{N} \binom{N}{i} \left((1-f)P_{FN}\right)^{i} \left(fP_{TN}\right)^{N-i}\right) \qquad (8.44)
\end{aligned}
$$

⇒ The percentage relative improvement in software reliability can be obtained by substituting Equation (8.44) in Equation (8.45)

$$
\begin{aligned}
RRI &= \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% \\
RRI &= \frac{R_{new} - (1-f)}{f} \times 100\% \qquad (8.45)
\end{aligned}
$$

- ***Safety***:
  The RBBV Pattern includes three design techniques: *diverse programming*, *fault detection and diagnosis (AT and Voting)*, and *recovery block*. According to the software requirements in the standard IEC 61508-3 [46], the recommendations for these techniques are shown in Tab. 8.10.

Table 8.10.: Recommendations for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Diverse programming	R	R	R	HR
Fault detection and diagnosis	–	R	HR	HR
Recovery block	R	R	R	R

According to the last table, the average recommendations of this pattern for the different safety integrity levels are shown in Tab. 8.11.

Table 8.11.: Recommendations of RBBV Pattern for safety integrity levels.

Pattern	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Recovery Block with Backup Voting Pattern	R	R	R	HR

For unsafe failures in this pattern, the RBBV may give invalid results that leads to an unsafe failure in one of the following two cases:

1. An alternate version passes the acceptance test with erroneous outcome at some stage i, $1 \leq i \leq N$, which means that the test fails the stages from 1 to $i-1$ and at stage i the outcome is erroneous and it passes the acceptance test.
2. The voter finds an erroneous majority among the stored values.

○ <u>*In the First Case*</u>:
Probability{a false positive case at stage i} $= P\{T\}^{i-1}P\{E \cap \overline{T}\}$
The probability that the *RBBV* passes the acceptance test with erroneous outcome at some stage i, $1 \leq i \leq N$

$$= \sum_{i=1}^{N} P\{T\}^{i-1}P\{E{\cap}\overline{T}\} \tag{8.46}$$

$$P\{E \cap T\} = P\{E\}P\{T|E\} = fP_{TN} \tag{8.47}$$

$$\begin{aligned} P\{E \cap \overline{T}\} &= P\{E\} - P\{E \cap T\} \\ P\{E \cap \overline{T}\} &= f - fP_{TN} = fP_{FP} \end{aligned} \tag{8.48}$$

By substituting Equation (8.48) and (8.37) in Equation (8.46), the probability that the $RBBV$ fails unsafely at some stage i

$$= \sum_{i=1}^{N} (fP_{TN} + (1-f)P_{FN})^{i-1} (fP_{FP}) \tag{8.49}$$

∘ <u>*In the Second Case*</u>:
The probability to find an erroneous majority between the backup values

$$\begin{aligned} &= \sum_{i=M}^{N} \binom{N}{i} (P\{erroneous\})^{i} (P\{correct\})^{N-i} \\ &= \sum_{i=M}^{N} \binom{N}{i} (fP_{TN})^{i} ((1-f)P_{FN})^{N-i} \end{aligned} \tag{8.50}$$

By adding Equation (8.49) and Equation (8.50) we obtain

$$\begin{aligned} P_{UF(new)} &= \left(\sum_{i=1}^{N} (fP_{TN} + (1-f)P_{FN})^{i-1} (fP_{FP})\right) \\ &\quad + \left(\sum_{i=M}^{N} \binom{N}{i} (fP_{TN})^{i} ((1-f)P_{FN})^{N-i}\right) \end{aligned} \tag{8.51}$$

⇒ The percentage relative safety improvement can be obtained by substituting Equation (8.51) in Equation (8.52)

$$\begin{aligned} RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\ RSI &= \left(1 - \frac{P_{UF(new)}}{f}\right) \times 100\% \end{aligned} \tag{8.52}$$

- ***Cost***:
  The cost can be divided into two parts.
  - <u>*Recurring Cost*</u>: Like the classical RB, this pattern runs the independent versions serially on a single hardware unit. Thus, there is no extra hardware requirement except the memory location to store the recovery checkpoint and the backup values.

- *Development Cost*: The development cost of the RBBV is approximately equivalent to the development cost of NVP and RB Pattern since it includes the development of independent and functionally equivalent N versions. The estimated practical cost, which can be conducted in the same way as for NVP, shows that the cost increases sublinearly with the number of components [33]. And each additional version costs about $75 - 80\%$ of a single version [68]. Moreover, this pattern includes extra cost for developing the acceptance test and the voting technique.

- ***Modifiability***:
There are four possible modifications
  1. *Modification of a single version*: It is possible to modify a single version either to remove a newly discovered fault, or to improve a poorly programmed function [12]. In this case, the initial specification remains without any modification, and the modification of this version is similar to the modification of single-version software following a standard fault removal procedure.
  2. *Modification of all member versions*: The reason for this modification is either to add a new functionality or to improve the overall performance of the N-version software [12]. In this case, the initial specification should be modified, and all the N versions should be modified and tested independently by independent teams to perform the new changes to the affected modules. In general, the modification of N-version software is remarkably more difficult than the modification of single-version software.
  3. *Modification of the acceptance test*: The acceptance test can be considered as independent module. Thus, it can be easily modified without any influence on the different versions.
  4. *Modification of the voter*: The separation of the voting module from the N versions and the acceptance test allows easy modifications or changes of the voting technique.

- ***Impact on Execution Time***:
Most of the time, only the primary version of the RBBV is executed, which keeps the run-time overhead of this pattern to a minimum value. In the worst case, all the alternate versions are executed without getting a reasonable result, followed by the voting method. Consequently, the execution time in the worst case will be

$$T_{WC} = N\left(t_{SV} + t_{AT}\right) + t_V \tag{8.53}$$

Where:
– t_{SV}: the execution time of a single version.
– t_{AT}: the execution time of the acceptance test.
– t_V : the execution time of the voter.
– t_{WC}: the worst case execution time of the RBBV methods.

Due to the weak acceptance test, it is more likely that more than one version are executed before offering a valid result, which increases the average execution time. Consequently, the execution time overhead of the RBBV in worst and average cases can be considered as the main disadvantage that makes this method unsuitable for real-time safety critical system with strict timing constraints.

Implementation:

1. While the RBBV can handle false negative cases, its ability to detect false positive, true positive, and true negative cases still depends on the quality of the acceptance test. Thus, the acceptance test should be carefully designed to allow high fault coverage.

2. The independent development of the required N versions and the level of diversity in these versions to avoid the common failures. In order to increase the level of diversity and the independence of the developed versions, the implementation issues presented in Section 8.2 should be taken into consideration as long as possible during the development process.

3. There are several voting techniques that can be used in this pattern to implement the voter component (see Section 8.4). The selection of the suitable technique depends on the data type, deviation in the outputs of the versions, type of agreement [63], output space cardinality size, functionality of the voter [75], reliability of different versions, and many other factors.

Consequences and Side Effects:
Similar to the classical RB, the main drawbacks of this pattern are the complexity of developing independent N versions in addition to the high development cost and the high dependency on the initial specification which may propagate faults to all versions.

Related Patterns:
To test the software versions for timeout condition and to provide sequence monitoring, the RBBV Pattern can be combined with the Watchdog Pattern.

9. Combination of Hardware and Software Patterns

9.1. Protected Single Channel Pattern (PSC)

Other Names:
–

Type:
Combination of Hardware and Software Pattern.

Abstract:
The Protected Single Channel Pattern [37] is a light weight pattern that is used to enhance the safety and reliability through the addition of checks and monitoring at different points in the channel. It is normally used to deal with transient faults.

Context:
Developing an embedded system with limited safety and reliability requirements in a situation where some level of redundancy is possible.

Problem:
How to deal with the transient faults to provide some level of safety and reliability to the embedded system in an inexpensive manner.

Pattern Structure:
The pattern structure is shown in Fig. 9.1. It contains two possible monitoring units: one for input data validation and one for actuation monitoring. Thus, there are two versions of this pattern; (open loop) when the input data validation unit is only used, and (closed loop) when the two units are used [37].

The function of each unit is as follows:

- *Actuation Channel*: (see the general pattern in Section 6.7).
- *Input Sensor(s)*: (see the general pattern in Section 6.7).

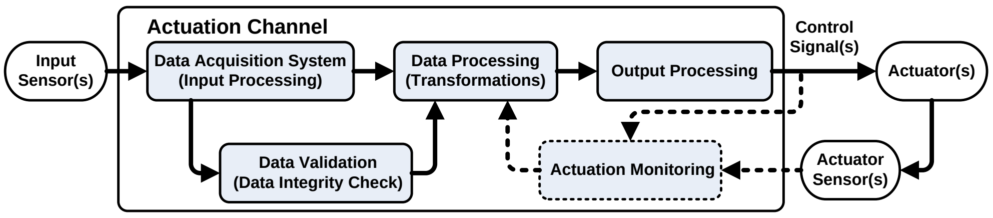


Figure 9.1.: Protected Single Channel Pattern.

- *Actuator(s)*: (see the general pattern in Section 6.7).
- *Data Acquisition (Input Processing)*: (see the general pattern in Section 6.7).
- *Data Processing (Transformation)*: (see the general pattern in Section 6.7).
- *Output Processing*: (see the general pattern in Section 6.7).
- *Actuator Sensor(s)*: The actuator sensors are used to get feed back signals from the output of the actuators to be used for the actuation monitoring.
- *Data Validation (Integrity Check)*: It may contain multiple components and provides checks on the input data and the system itself during the executing of the desired algorithm. The data integrity check may contain range checks or correctness check by computing parity or CRC checks.
- *Actuator Monitoring*: It is an optional part that is used for closed loop version of this pattern. It provides a monitoring to the output of the channel, such as checking the output commands for validity before delivering this command to the actuators. It can also check the output actuators using separate sensors by getting feedback values from the actuators and comparing these values with the previously generated control signals. Consequently, it can use this information to reconfigure the output processing component to overcome the transient faults when it is possible.

Implication:
This section shows the implication of this pattern relative to the basic system. We will use the following notations for this pattern:

– f	:	The probability of failure in the actuation channel ($f = 1 - R$).
– P_{tf}	:	The probability that a given fault is a transient fault.
– C_{tf}	:	The coverage factor of this pattern which is equal to the ratio of the discovered transient faults by this pattern to the total possible transient faults.

- ***Reliability***:
  The remaining failures after using this pattern include the other types and the undiscovered transient failures, with probability $= (1 - C_{tf}P_{tf})\, f$.

  $$R_{new} = 1 - (1 - C_{tf}P_{tf})\, f \tag{9.1}$$

  $\Rightarrow$ The percentage relative improvement in reliability is

  $$\begin{aligned} RRI &= \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% \\ RRI &= \frac{1 - (1 - C_{tf}P_{tf})\, f - (1 - f)}{1 - (1 - f)} \times 100\% \\ RRI &= C_{tf}P_{tf} \times 100\% \end{aligned} \tag{9.2}$$

- ***Safety***:
  This pattern includes a single technique which can be classified as *failure detection by online monitoring.* According to the hardware requirements in the standard IEC 61508-2 [46], the recommendations for this technique are shown in Tab. 9.1.

Table 9.1.: Recommendations of PSC Pattern for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Protected Single Channel Pattern (failure detection by online monitoring)	HR	HR	HR	HR

The protected single channel pattern can only cover the transient faults to provide some level of safety. But it can not continue to operate safely in the presence of persistent faults. Therefore, *it is not applicable to work alone* for any situation that requires any special safety integrity. It should be integrated with another safety technique in the presence of immediate fail-safe state to be used for light safety-critical applications.

To compute RSI:

$$P_{UF(old)} = 1 - R_{AC} = 1 - R = f \tag{9.3}$$

The probability of unsafe failure after using this pattern:

$$P_{UF(new)} = (1 - C_{tf}P_{tf})\, f \tag{9.4}$$

$\Rightarrow$ The percentage relative safety improvement is

$$\begin{aligned} RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\ RSI &= \frac{(1 - C_{tf}P_{tf})\,f - f}{0 - f} \times 100\% \\ RSI &= C_{tf}P_{tf} \times 100\% \qquad (9.5) \end{aligned}$$

Equation (9.5) and Equation (9.2) show that the relative safety improvement is equal to the relative reliability improvement.

- ***Cost***:
  The extra cost in this pattern depends on the method of implementation, and can be classified into two parts:
  - *Recurring Cost*: It includes the cost of extra sensors, data validation unit if it is implemented as hardware component, and the cost of memory locations for data redundancy such as(CRC).
  - *Development Cost*: The cost for developing a software procedure to perform some part of data validation and checking.

  In general, the cost for this pattern is small comparing to the other patterns that include high level of redundancy.
- ***Modifiability***:
  It is possible to modify this pattern by adding extra module for further data validation or actuator monitoring, but this involves the extra hardware units which will increase the recurring cost. On the other hand, the validation and checking algorithm must be modified to take into consideration the new components and the new checking algorithms.
- ***Impact on Execution Time***:
  The influence of this pattern on the executing time depends on the implementation:
  - If the validation checks are performed by hardware, then the extra components are working in parallel with the basic channel which does not affect the basic system in the normal execution.
  - In the case of software implementation, the increase in the execution time depends on the required time to execute the extra algorithm comparing to the original software.

Implementation:

- In this pattern, there are two versions to be implemented: either (open loop) with only data integrity checking unit, or (close loop) that includes additional actuation monitoring unit.
- The designer should determine the input data to monitor and he must have a semantic knowledge about the data, which can be used for out of range value checking.
- There are two options to provide the data checking: either by software or by hardware.
- When there is no time constraint in the application, the designer should try to use software checking, to reduce the recurring cost, especially in products with large quantities.
- For data integrity, redundant storage unit can be used. For example, to provide checksum for stored data that can be computed at runtime and can be compared with the stored value in the storage unit.
- It is good choice to store the redundant data in inverted format to detect stuck-at RAM fault.
- When it is possible to have an immediate fail-safe state, it should be used in the design to increase the safety level. The existence of a fail-safe state gives the data integrity and the actuator monitoring components the capability to switch the system into the fail-safe state in the presence of persistent fault.

Consequences and Side Effects:
The main drawback of this pattern is that it is not appropriate for dealing with persistent faults, since it can not continue to operate safely, and sometime leads to the loss of the entire system.

Related Patterns:
To give a high level of integrity, additional patterns can be used beside this pattern to deal with the random and systematic persistent faults. Such as homogeneous duplex, heterogeneous redundancy, and triple modular redundancy pattern.

9.2. 3-Level Safety Monitoring Pattern (3-LSM)

Other Names:
E-GAS.

Type:
Combination of Hardware and Software Pattern.

Abstract:
The 3-Level Safety Monitoring Pattern is considered as a combination of the Monitor-Actuator Pattern and the Watchdog Pattern to be suitable for the applications that require a continuous safety monitoring and include a fail-safe state without high hardware redundancy. It consists of a single hardware channel that includes 3 levels: actuation (function), monitoring, and control level. The function level executes the subprogram for carrying out the intended functionality, while the monitoring level monitors the first level, and the control level controls the monitoring level and the entire hardware channel. Furthermore, a watchdog, which communicates via periodic messages with the control level, is used to reset the system into its fail-safe state in the case of failure.

Context:
Developing an embedded system with a given fail-safe state or a corrective action, and without hardware redundancy.

Problem:
How to improve the safety of an embedded system at low reasonable cost, in the presence of failure in a system with a fail-safe state. Furthermore, how to continue providing the required safety level and to ensure that the system does no injure or harm, when there is any deviation in the output of the actuators from the commanded set point.

Pattern Structure:
The 3-Level Safety Monitoring was proposed by ***Robert Bosch GmbH*** to be used in the ***E-Gas*** unit as a method for management and controlling the drive power of a motor vehicle [17]. It consists of a single hardware channel to perform three computing elements: Actuation, monitoring and control element. This pattern provides higher safety monitoring than the classical watchdog pattern since it includes a functional monitoring element besides the sequence control provided in the classical watchdog pattern. Nevertheless, the fault coverage in this patter is less than in the Monitor-Actuator and the Sanity Check Pattern since it includes a single hardware channel instead of two.

The structure of this pattern is shown in Fig. 9.2, where the function of each unit is as follows:

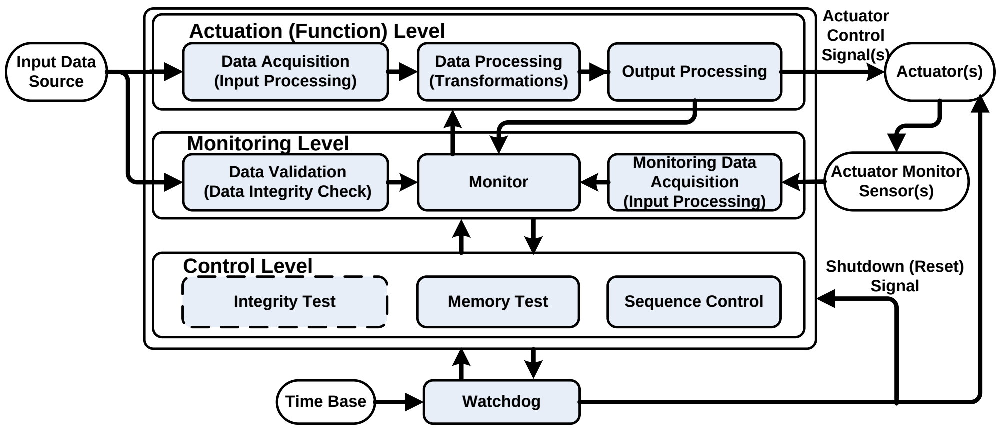

Actuation (Function) Level
Input Data Source
Data Acquisition (Input Processing)
Data Processing (Transformations)
Output Processing
Actuator Control Signal(s)
Actuator(s)
Monitoring Level
Data Validation (Data Integrity Check)
Monitor
Monitoring Data Acquisition (Input Processing)
Actuator Monitor Sensor(s)
Control Level
Integrity Test
Memory Test
Sequence Control
Shutdown (Reset) Signal
Time Base
Watchdog

Figure 9.2.: 3-Level Safety Monitoring Pattern

- *Actuation Module (Function Level)*: It is the main computing element which is dedicated to perform the basic functionality. It performs some tasks in the complete system by taking an input data from an external input data source like a sensor, and performs some transformation on this data and then it uses the results to generate suitable control signals.
- *Monitoring Module (Level)*: It monitors the correct or defective operation of the first actuation module through a comparison of the processing results, input data and the data from the actuator's sensors. In the case of great difference between the desired value and the measured value, this module forces the actuation channel to switch into its fail-safe state.
- *Control Module (Level)*: This module monitors and provides a control level to the monitoring level and to the complete hardware channel. It cooperates with an external watchdog to achieve the sequence control functionality for the second level.
- *Input Data Source*: (see the general pattern in Section 6.7).
- *Actuator(s)*: (see the general pattern in Section 6.7).
- *Data Acquisition (Input Processing)*: (see the general pattern in Section 6.7).
- *Data Processing (Transformation)*: (see the general pattern in Section 6.7).

- *Output Processing*: (see the general pattern in Section 6.7).
- *Data Validation (Data Integrity Check)*: It provides a check on the input data during the executing of the desired algorithm to ensure that the input data is valid and in the safe boundaries.
- *Actuator Monitor Sensor(s)*: It represents a source of information to the monitoring module, where it is used to get feedback signals from the output of the actuators to provide a monitoring to the actuation level.
- *Monitoring Data Acquisition*: It collects the raw data from the actuator sensors and may convert the data to another form, and then it transfers the data to the monitor component.
- *Monitor*: The monitor takes information about the status of the actuator and compares it with the provided input commands and with the actuator control signal computed in the first level. The result of comparison is used to ensure that the actuation output is approximately correct and within a given range. If the result of the comparison shows that the output of actuation channel is totally incorrect and may affect the system safety, the monitor will generate a correction command or shutdown signal to switch the system into its fail-safe state.
- *Watchdog*: It is used to provide a sequence control to the monitoring level and to the entire actuation channel. The functionality of the watchdog varies from: a simple liveness messages (strokes) received from the main hardware of the actuation channel in a predefined timeframe, to a complex query messages between the watchdog and the sequence control module. In the last case, the watchdog transmits a query determined by a random number generator to the sequence control. The sequence control invokes some tests and constructs the result signal which is sent back to watchdog. The watchdog compares the received result with the original transmitted query. If the watchdog detects a deviation, then it considers this situation as a fault in the hardware channel, and then it issues a shutdown signal to switch the main channel and the actuators to the fail-safe state.
- *Sequence Control*: The sequence control module receives the query transmitted from the watchdog, and then evaluates this message and determines the subprograms or tests that should be executed to reply the query. When all the selected subprograms and tests have been executed, then the result message is constructed and transmitted back to the watchdog.
- *Memory Test*: This module includes read/write *RAM* test and read *ROM* test to insure the correctness and integrity of the internal memory.

- *Integrity Test (Optional)*: This is an optional component that is invoked by the sequence control module to run built in tests to verify all or a portion of the internal functionality of the main actuation channel.
- *Time Base*: It is an independent timing source to drive the watchdog. This source is separate from the one used to drive the actuation channel.

Implication:
This section shows the implication of this pattern relative to the basic system.

- ***Reliability***:
  As shown previously, this pattern is used to provide a safety monitoring to the actuation channel, to identify the faults and then to generate the shutdown or reset signal to force the actuation channel entering the fail-safe state. On the other hand, this pattern does not change the failure rate of the system ($\lambda_{new} = \lambda_{old}$), and the hardware actuation channel cannot continue to function when a fault is detected. Therefore, this pattern does not improve the reliability of the system ($R_{new} = R_{old}$).
  ⇒ The percentage relative improvement in reliability is
  $$RRI = \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% = 0\% \tag{9.6}$$
- ***Safety***:
  The 3-LSM Pattern includes two design techniques: *program sequence monitoring* and *fault detection and diagnoses through safety monitoring.* According to the software requirements in the standard IEC 61508-3 [46], the recommendations for these techniques are shown in Tab. 9.2.

Table 9.2.: Recommendations for safety integrity levels.

Techniques	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Program sequence monitoring	HR	HR	HR	HR
Fault detection and diagnoses	–	R	HR	HR

According to the last table, the average recommendations of this pattern for the different safety integrity levels are shown in Tab. 9.3.

Table 9.3.: Recommendations of 3-LSM Pattern for safety integrity levels.

Pattern	**SIL1**	**SIL2**	**SIL3**	**SIL4**
3-Level Safety Monitoring Pattern	R	MR	HR	HR

To compute RSI, let us have the following notations for this pattern:

– R_{AC} :	The reliability of the actuation channel.
– R_{WD} :	The reliability of the watchdog component.
– C_{ML} :	The coverage factor of the monitoring level which is equal to the probability that an unsafe failure in the actuation module is detected by the monitoring module.
– C_{WD} :	The coverage factor of the watchdog and control level which is equal to the probability that undetected failure, either in monitoring module or in the hardware actuation channel, is detected by the watchdog and the control level.
– E_{ML} :	The event that a failure is detected by the monitoring level.
– E_{WD} :	The event that a failure is detected by the watchdog and control level.

The probability of failure in the actuation channel is equal to the failure in a simple channel

$$P_{UF(old)} = 1 - R_{AC} = 1 - R \tag{9.7}$$

The failures in the actuation channel can be divided into two groups:

1. Safe failures detected by the monitoring module or the control module with the watchdog.
2. Unsafe failures those are undetected neither by the monitoring module nor by the control module with the watchdog.

$*$ The probability that a failure is detected by the monitoring module

$$= (1 - R_{AC})\, C_{ML} \tag{9.8}$$

$*$ The probability that a failure is not detected by the monitoring module

$$= (1 - R_{AC})\,(1 - C_{ML}) \tag{9.9}$$

$*$ The probability that a failure is not detected by the monitoring level and then detected by the watchdog

$$= (1 - R_{AC})\,(1 - C_{ML})\, R_{WD} C_{WD} \tag{9.10}$$

$*$ The probability that a failure is not detected neither by the monitoring level nor by the watchdog

$$= (1 - R_{AC})\,(1 - C_{ML})\,(1 - R_{WD} C_{WD}) \tag{9.11}$$

Therefore, the probability of unsafe failure in this pattern is

$$P_{UF(new)} = (1 - R)\,(1 - C_{ML})\,(1 - R_{WD}C_{WD}) \tag{9.12}$$

If we assume that the watchdog was carefully designed ($R_{WD} \approx 1$) then

$$P_{UF(new)} \approx (1 - R)\,(1 - C_{ML})\,(1 - C_{WD}) \tag{9.13}$$

$\Rightarrow$ The percentage relative safety improvement is

$$\begin{aligned}
RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\
RSI &= \frac{(1 - R)\,(1 - C_{ML})\,(1 - C_{WD}) - (1 - R)}{0 - (1 - R)} \times 100\% \\
RSI &= (C_{ML} + C_{WD} - C_{ML}C_{WD}) \times 100\% \qquad (9.14)
\end{aligned}$$

If we define the coverage factor for the 3-Level Safety Monitoring Pattern (C_{3LSM}) to be the probability that a failure is detected either by the monitoring level or by the control level and the watchdog, then this coverage factor can be calculated as follows:

$$\begin{aligned}
C_{3LSM} &= P\{E_{ML} \cup E_{WD}\} \\
C_{3LSM} &= P\{E_{ML}\} + P\{E_{WD}\} - P\{E_{ML}\}P\{E_{WD}\} \\
C_{3LSM} &= (C_{ML} + C_{WD} - C_{ML}C_{WD}) \qquad (9.15)
\end{aligned}$$

$\Rightarrow$ The percentage relative safety improvement is

$$RSI = C_{3LSM} \times 100\% \tag{9.16}$$

It is clear from Equation (9.16) that the safety improvement depends on the coverage factor of this pattern, which in turn depends on the coverage factor of the monitoring module and the coverage factor of the watchdog. The monitoring module in this pattern is not able to detect hardware failure in the main hardware channel since it is executed by the same hardware channel as the functional level. Therefore, the coverage factor and the safety improvement in this pattern is less than in the Monitor-Actuator and the Sanity-Check Pattern. Nevertheless, the safety improvement is greater than the classical watchdog pattern.

- ***Cost***:
  The low-cost of this pattern can be classified into two parts:
  - *Recurring Cost*: It includes the cost of the watchdog component, independent timing source and extra sensors.

  - *Development Cost*: It include the cost for developing the monitoring module, control level and periodic tests.

- ***Modifiability***:
  The modification of the actuation level to add extra features is considered to be an easy step that involves the following:
  - The modification of the monitoring module to provide a monitoring to the new features, and this will include additional new sensors, modification of the monitoring data acquisition and modification of the monitor in order to take the new features into consideration.
  - The modification of the control module to include extra tests.

- ***Impact on Execution Time***:
  As shown in the pattern structure, the three levels are executed on the same hardware channel; thus, the total execution time will be affected by the time to execute the actuation level, the monitoring module, and the periodic tests in the control level. Therefore, the additional execution time will depend on the complexity of the monitoring and control level.

Implementation:

- In order to decrease the recurring cost, two different types of sensors can be used for implementation: high sensitivity sensors for the basic actuation level, and low-cost and low-sensitivity sensors for the monitoring level.
- If low-sensitive sensors are used for the monitoring level, then a simple algorithm should be used to perform the required broad range comparison.
- Based on the level of complexity, some factors like: time lag, measurement and computational accuracy, and data formatting errors, might affect the comparison algorithm in the monitoring level. Therefore, these factors should be taken into consideration in the implementation of this pattern.

Consequences and Side Effects:
The main drawback for this pattern is that it includes a single hardware channel; thus, it can not be used to tolerate hardware failures in applications with high reliability and availability requirements.

Related Patterns:
The Monitor-Actuator Pattern is the most related pattern to the 3-Level Safety Monitoring Pattern, since it represent a special implementation when the monitoring and control level are carried out in a separate hardware channel with the addition of a watchdog.

10. Conclusion

This chapter summarizes the contributions of this thesis and draws some conclusions about this research. Finally, it outlines some possible directions for future work.

10.1. Summary

In this thesis, we developed an approach for the adoption of the design pattern concept for safety-critical embedded systems design. A catalog of design patterns was constructed to provide a set of well-known, structured, and documented design solutions for safety-critical embedded systems.

The representation of design solutions as design patterns serves two advantages. Firstly, it can simplify communication between practitioners by providing a good way for documentation of proven design techniques; and secondly, it support and help designers and system architects choose a suitable solution for a recurring design problem among available collection of successful solutions.

Unlike traditional computer system, safety-critical embedded systems have special functional and non-functional requirements that should be met by the available design methods. Consequently, these requirements should be included when the design solutions are presented as design patterns.

To overcome the lack of consideration of non-functional side effects and potential consequences in the available pattern representations, Chapter 3 presented an effective pattern representation template that focuses on the implication and side effects on the non functional requirements. Comparing to the traditional pattern representations, this template includes the essential elements in addition to a section for the implications on non-function requirements that include safety, reliability, modifiability, cost and execution time.

Among the different requirements for safety-critical systems, this thesis focused on safety and reliability as they have direct effects on the users and the environment. These two requirements are used as major factors in the differentiation between available patterns. Chapter 4 described reliability and safety assessment method that can be employed to facilitate the comparison possess of design patterns for their suitability for a specific design problem.

The reliability assessment method includes a reliability metric which describes the percentage reliability improvement relatively to a basic system. This relative computation overcomes the abstraction problem of design patterns and gives an indication about the possible improvement that can be achieved when using the pattern under consideration. Similarly, the safety assessment method includes a safety metric based on the computation of the relative safety improvement based on the possible reduction in the probability of unsafe failures. Moreover, the safety assessment includes a system of safety recommendations derived from the safety integrity levels, presented in the IEC 61508 standard, and the recommendations for these levels. The combination of these two parts in the safety assessment reflects the importance of the addressed design pattern and the possible severity of the target applications.

The question that arises is how to apply these abstract metrics. In this thesis, we presented two approaches to use the developed safety and reliability metrics in the assessment process of design patterns:

- Through the employment of a mathematical modelling, as done in our catalog of patterns.
- By using simulation techniques.

Chapter 5 clarified how to carry out a simulation method to demonstrate the presented assessment methods. It described a case study that includes a Monte Carlo based simulation to assess four software design patterns. Furthermore, it illustrated how to conduct a comparison between patterns, and how to investigate the main parameters that affects their reliability and safety.

Chapter 6 described the architecture of the software version of the constructed catalog of design patterns. It detailed the main modules and the data model used to implement this catalog. The main features provided in this catalog can be summarized in: easy to use, well-documented patterns, interactive navigation and browsing, simulation of software patterns, and a decision support for users.

The selection of a suitable design pattern for a given design problem is considered to be a tedious and unstructured process. Therefore, the decision support developed in our catalog aims to help users to overcome decision making problems. The developed approach involved the following:

- Provision of a set of possible pattern combinations as a set of solutions.
- Categorizing these solutions into groups based on the problems that can be solved.
- Construction of decision trees that associate the applicable solutions for each design problem with the requirements for these solutions.

The decision support was developed in our tool as a search wizard which exploits the constructed decision trees to assist users in the selection process.

The last part of this thesis, Chapter 7, 8 and 9, presented the design patterns that form our catalog. The collected patterns included fifteen patterns which are divided into three groups based on the type of patterns: hardware, software, and a combination of hardware and software patterns. We think that this division is sufficient for our catalog since it classifies all patterns that share the same design concepts and solve related problems in a same category.

Summarizing, we believe that applying the concept of design patterns to the field of safety-critical embedded systems is very promising in the use of proven solutions. The integration of non-functional requirements in the proposed representation aims to give an indication about the possible impacts and side effects of a considered design pattern and to help in the general assessment of design patterns.

While the proposed assessment method covers only safety and reliability, there are many other factors that should be taken into consideration to achieve a comprehensive comparison. These factors include, among others: costs, time overhead and maintainability.

On the other hand, the main criticisms of our approach can be derived from the general criticism of the concept of design patterns. It can be claimed that the non-functional requirements are highly dependant on the implementation, and the abstract nature of design patterns does not give complete information about the final system.

While we do not argue that design patterns give partial information, we think that the introduction of this abstract information at the early stages of a design can be of great benefit to the design process. Moreover, we have to keep in mind that the use of the recommendations that are presented in our patterns and in the safety assessment method does not guarantee the satisfaction of the required safety integrity in the final design. In contrary, the standard should be followed throughout the entire life-cycle in order to get a certificate for a specific safety integrity level.

10.2. Future Work

The research presented in this thesis introduces some possible directions for future work.

- The presented catalog only includes fifteen design patterns that represent common design solutions. Thus, this catalog is applicable to be extended to include more patterns. Other design techniques that address safety-critical embedded systems can be represented as pattern and later be included in this catalog.

- The developed simulation module in our catalog only covered five software patterns for the conducted case study. Therefore, the construction of a comprehensive simulator that provides reliability and safety simulation for all design patterns would be desirable and useful for comparison and assessment.
- In our pattern representation template, we have included five non-functional requirements to cover general safety-critical applications. Further research is needed to modify this template to cover other specific requirements for other specific applications. For example, in many safety-critical application areas, system and information security from unauthorized access represents a property that has to be considered in these applications. Therefore, these specific requirements are applicable to be integrated in this template.
- The transfer of the concept of design pattern to specific applications of safety-critical embedded systems, like real-time safety-critical embedded systems, in order to construct application-oriented catalogs might be interesting and useful.

Finally, we think that the integration of the suggested research approaches with the catalog developed in this thesis would be more helpful for designers to select appropriate solutions for their design problems.

A. Decision Trees

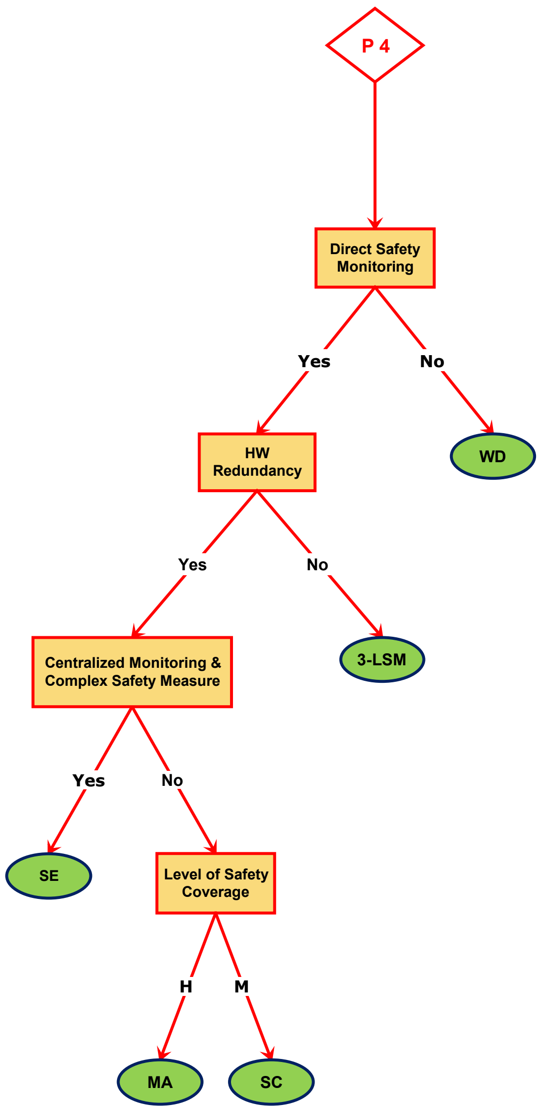


Figure A.1.: The decision tree for the safety monitoring problem.

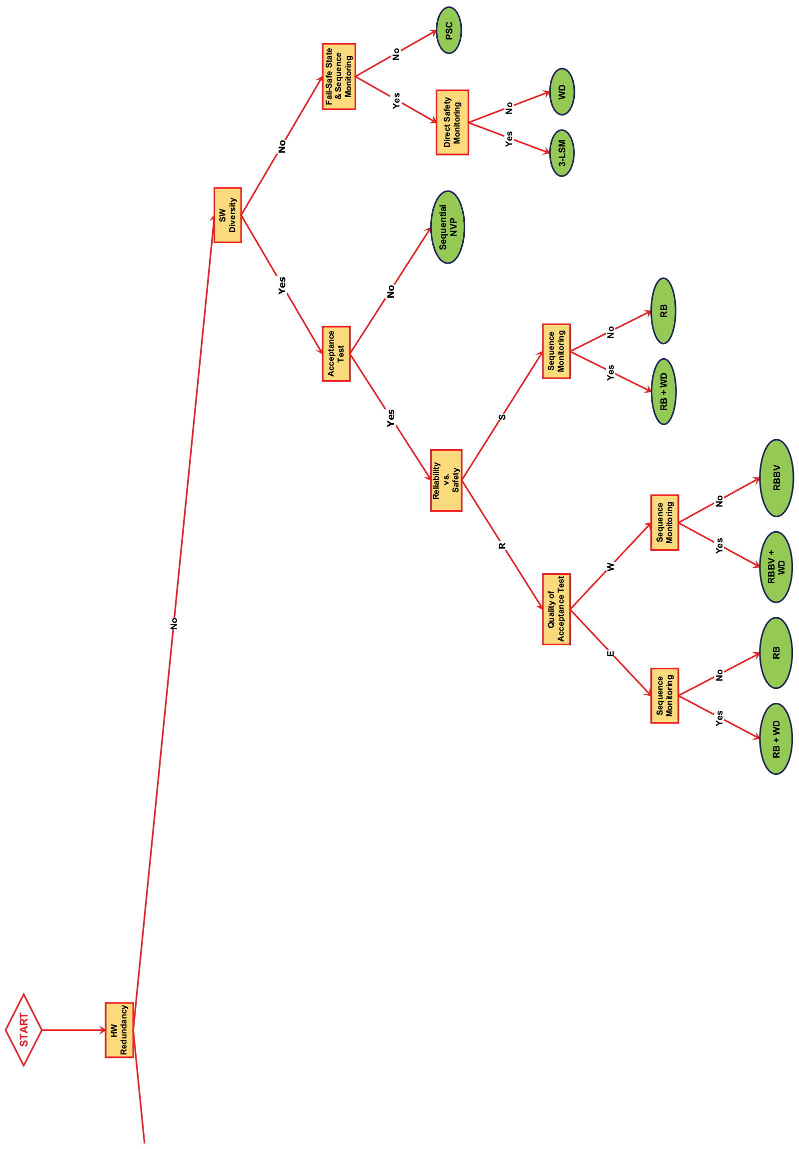


Figure A.2.: The decision tree for the general design problem (Part 1).

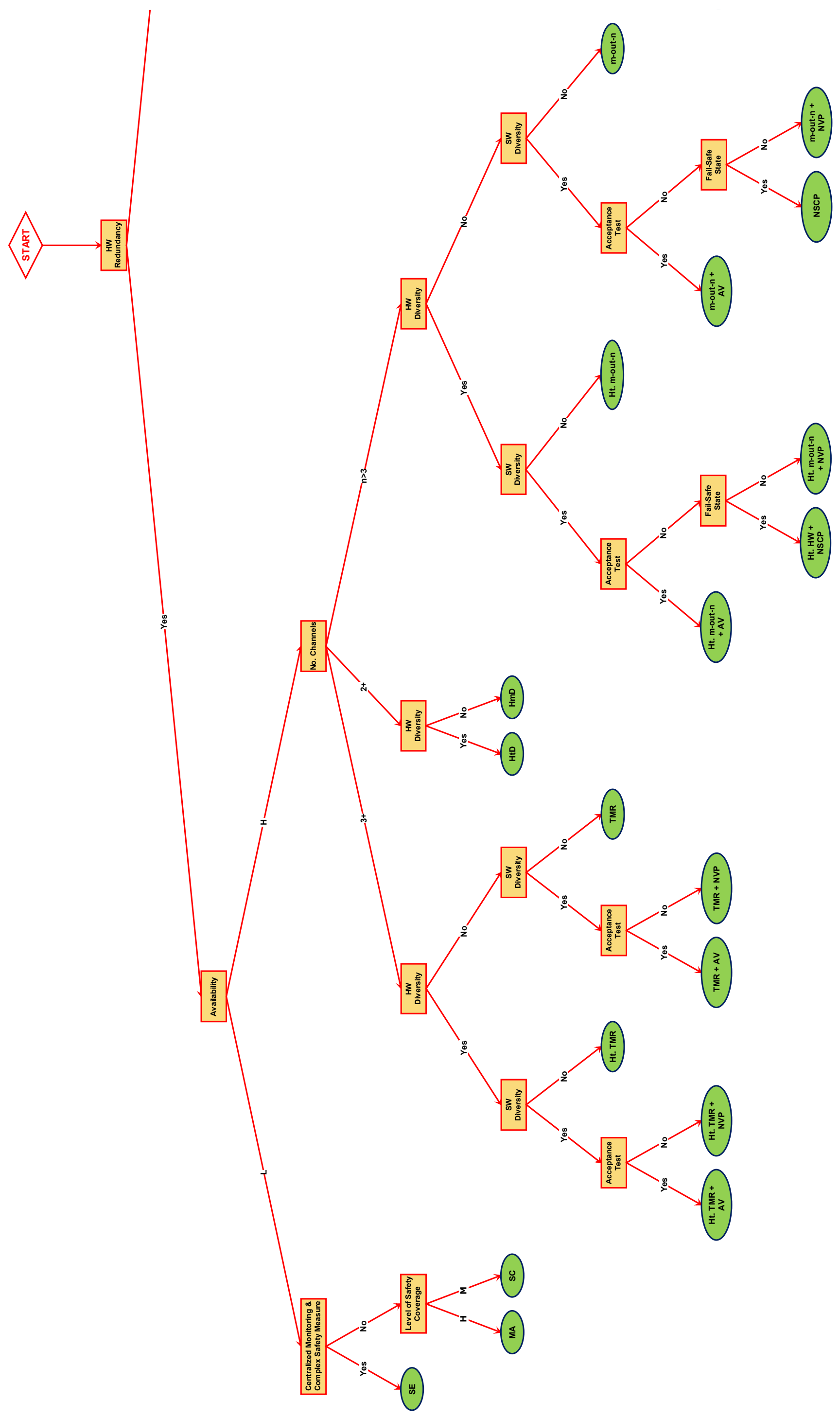


Figure A.3.: The decision tree for the general design problem (Part 2).

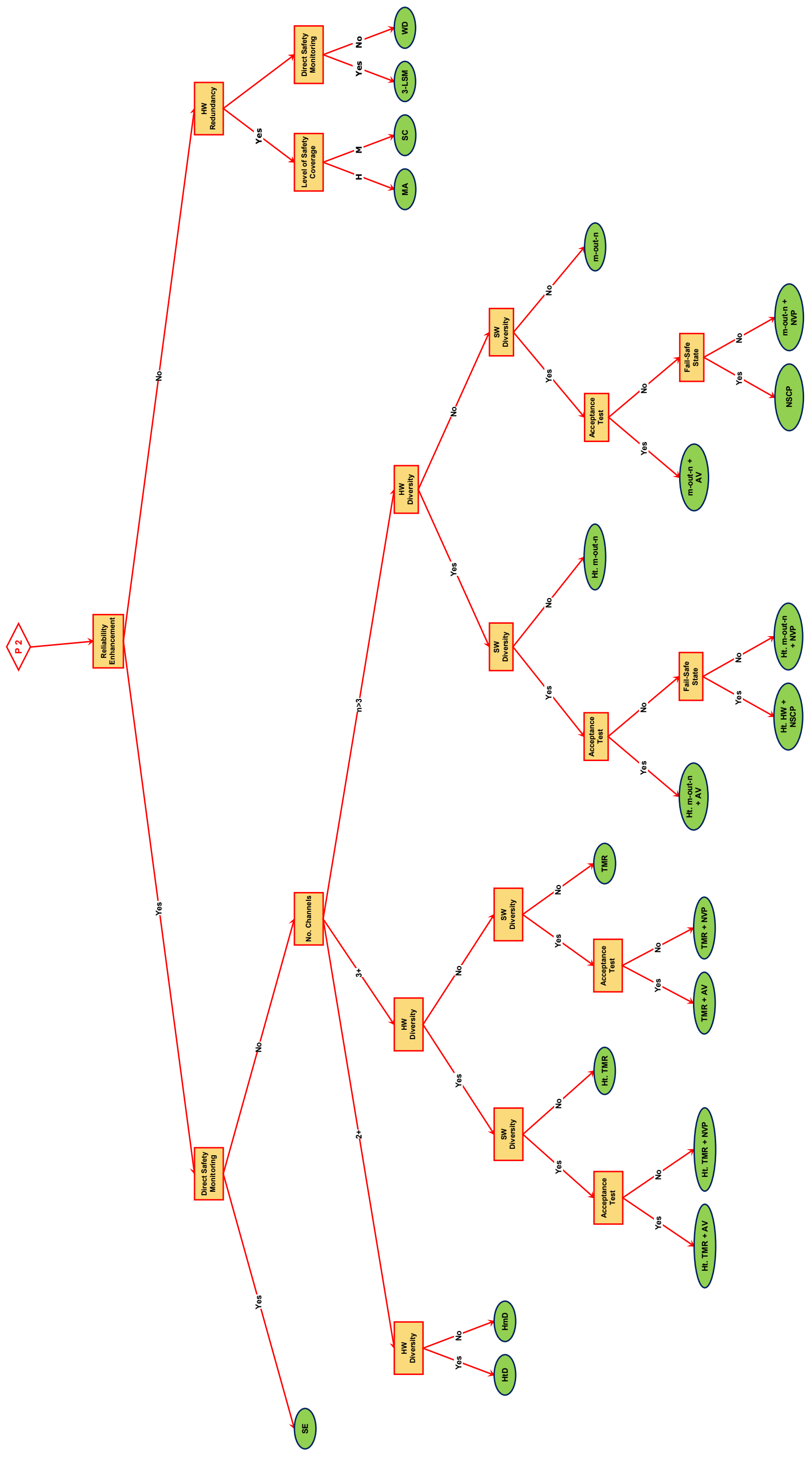


Figure A.4.: The decision tree for the hardware random faults problem.

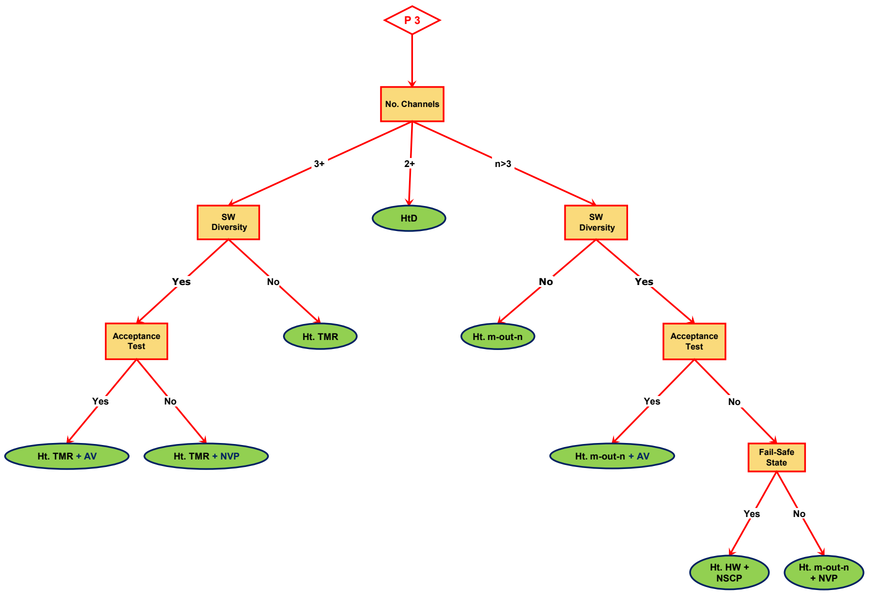


Figure A.5.: The decision tree for the hardware systematic faults problem.

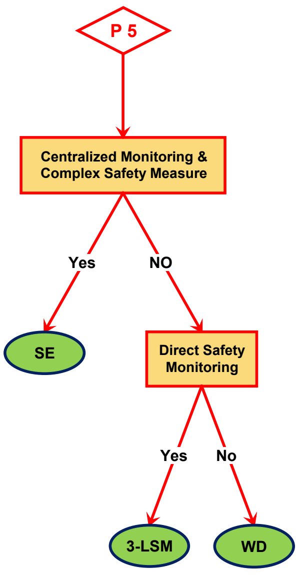


Figure A.6.: The decision tree for the sequence monitoring problem.

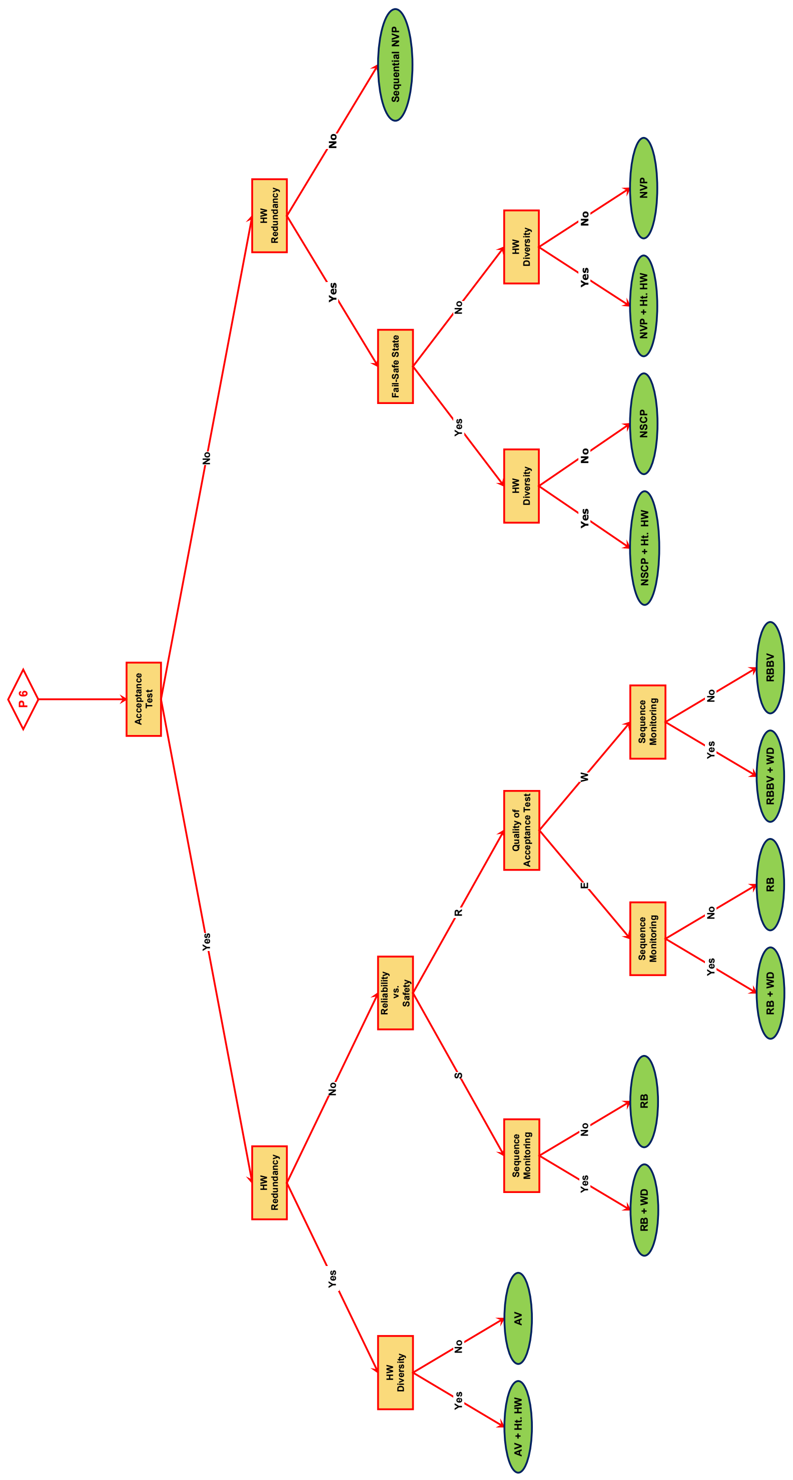


Figure A.7.: The decision tree for the software faults problem.

Bibliography

[1] S. Adams. Functionality ala carte. *Pattern Languages of Program Design*, pages 7–8, 1995.

[2] C. Alexander. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, New York, 1977.

[3] T. Anderson and R. Kerr. Recovery blocks in action: A system supporting high reliability. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 447–457, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[4] A. Armoush, E. Beckschulze, and S. Kowalewski. Safety assessment of design patterns for safety-critical embedded systems. In *35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009).* IEEE CS, Aug. 2009.

[5] A. Armoush and S. Kowalewski. Safety recommendations for safety-critical design patterns. In *Design of Dependable Critical Systems (DDCS) In the framework of the SAFECOMP2009*, 2009.

[6] A. Armoush, F. Salewski, and S. Kowalewski. Effective pattern representation for safety critical embedded systems. In *2008 International Conference on Computer Science and Software Engineering*, volume 4, pages 91–97. IEEE CS, Dec. 2008.

[7] A. Armoush, F. Salewski, and S. Kowalewski. A hybrid fault tolerance method for recovery block with a weak acceptance test. In *The 5th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC 2008)*, volume 1, pages 484–491. IEEE CS, Dec. 2008.

[8] A. Armoush, F. Salewski, and S. Kowalewski. Recovery block with backup voting: A new pattern with extended representation for safety critical embedded systems. In *11th International Conference on Information Technology (ICIT 2008)*, pages 232–237. IEEE CS, Dec. 2008.

[9] A. Armoush, F. Salewski, and S. Kowalewski. Design pattern representation for safety-critical embedded systems. *Journal of Software Engineering and Applications (JSEA)*, 2(1):1–12, April 2009.

[10] A. Athavale. Performance evaluation of hybrid voting schemes. Master's thesis, North Carolina State University, Department of Computer Science, 1989.

[11] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[12] A. Avizienis. *The Methodology of N-version Programming*, chapter Software Fault Tolerance, pages 23–46. Wiley, New York, 1995.

[13] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *IEEE Computer Software and Applications Conference 77*, pages 149–155. IEEE Press, 1977.

[14] A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability. Technical Report N01145, LAAS-CNRS, April 2001.

[15] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[16] K. Beck and W. Cunningham. Using pattern langugages for object-oriented programs. In *Position Paper for the Specification and Design for Object-Oriented Programming Workshop, The 3rd Conference on Object-Oriented Programming Systems, Languages and Applications*, Orlando, USA, 1987.

[17] F. Bederna and T. Zeller. United States Patent No. 5,880,568: Method and arrangement for controlling the drive unit of a vehicle, March 1999.

[18] A. Berger. *Embedded systems design: an introduction to processes, tools, and techniques.* Focal Press, 2^{nd} edition, 2002.

[19] A. Birolini. *Reliability Engineering, Theory and Practice.* Springer, 5th edition, 2007.

[20] F. Bitsch. Safety patterns - the key to formal specification of safety requirements. In *the 20th International Conference on Computer Safety, Reliability and Security (SAFECOMP '01)*, pages 176–189, London, UK, 2001. Springer-Verlag.

[21] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 4^{th} edition, 2009.

[22] F. Buschmann. *The Master-Slave Pattern.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[23] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: a system of patterns.* John Wiley & Sons, Inc. New York, 1996.

[24] G. Buttazzo. Research trends in real-time computing for embedded systems. *SIGBED Rev.*, 3(3):1–10, 2006.

[25] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, "Highlights from Twenty-Five Years"*, page 113, June 27–30 1995.

[26] P. P.-S. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, 1976.

[27] J. Cleland-Huang and D. Schmelzer. Dynamically tracing non-functional requirements through design pattern invariants. In *Workshop on Traceability in Emerging Forms of Software Engineering, in conjunction with IEEE International Conference on Automated Software Engineering*, October 2003.

[28] P. Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152–159, 1992.

[29] J. O. Coplien. *Advanced C++ programming styles and idioms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1992.

[30] J. O. Coplien. Idioms and patterns as architectural literature. *IEEE Softw.*, 14(1):36–42, 1997.

[31] R. Damaševičius, G. Majauskas, and V. Štuikys. Application of design patterns for hardware design. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 48–53, New York, NY, USA, 2003. ACM.

[32] R. Damaševičius and V. Štuikys. Application of UML for hardware design based on design process model. In *ASP-DAC '04: Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, pages 244–249, Piscataway, NJ, USA, 2004. IEEE Press.

[33] F. Daniels, K. Kim, and M. Vouk. The reliable hybrid pattern: a generalized software fault tolerant design pattern. In *Conference PloP'97*, pages 1–9, 1997.

[34] T. DeLong, D. Smith, and B. Johnson. Dependability metrics to assess safety-critical systems. *IEEE Transactions on Reliability*, 54(3):498–505, 2005.

[35] DOD. *MIL-STD 882D–Standard Practice for System Safety.* US Department of Defense (DOD), 2000.

[36] B. P. Douglass. *Doing Hard Time: Developing Real-Time System with UML, Objects, Frameworks, and Pattern.* Addison-Wesley, New York, 1999.

[37] B. P. Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems.* Addison-Wesley, New York, Boston, MA, USA, 2002.

[38] W. R. Dunn. *Practical Design of Safety-Critical Computer Systems.* Reliability Press, 2002.

[39] W. R. Dunn. Designing safety-critical computer systems. *Computer*, 36(11):40–46, 2003.

[40] J. Fletcher and J. Cleland-Huang. Softgoal traceability patterns. In *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 363–374, Washington, DC, USA, 2006. IEEE Computer Society.

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Boston, MA, USA, 1997.

[42] A. Grinin. Development of a catalog of design patterns for safety-critical embedded systems. Diploma thesis, RWTH Aachen University, 2009.

[43] D. Gross and E. Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6(1):18–36, 2001.

[44] L. Grunpke. Transformational patterns for the improvement of safety properties in architectural specification. In *Proceeding of the Viking PLoPO3*, Bergen, Norway, 2003.

[45] R. Hanmer. *Patterns for Fault Tolerant Software.* Wiley Publishing, 2007.

[46] IEC61508. *Functional safety for electrical / electronic / programmable electronic safety-related systems.* International Electrotechnical Comission, 1998.

[47] U. Isaksen, J. P. Bowen, and N. Nissanke. System and software safety in critical systems. Technical Report RUCS/97/TR/062/A, The University of Reading, UK, 1999.

[48] ISO. *ISO/WD 26262 - Road vehicles - Functional Safety.* International Organization for Standardization, working draft (2007).

[49] K. Kanoun, M. Kaaniche, C. Beounes, J.-C. Laprie, and J. Arlat. Reliability growth of fault-tolerant software. *IEEE Transactions on Reliability*, 42(2):205–219, 1993.

[50] J. Kelly, T. McVittie, and W. Yamamoto. Implementing design diversity to achieve fault tolerance. *IEEE Transactions on Software*, 8(4):61–71, 1991.

[51] T. P. Kelly. *Arguing Safety - A Systematic Approach to Managing Safety Cases.* PhD thesis, Department of Computer Science - University of York, 1998.

[52] K. Kim. Distributed execution of recovery block: An approach to uniform treatment of hardware and software faults. In *Proc. Fourth IEEE Int'l Conf. Distributed Computing Systems (ICDCS)*, pages 526–532, San Francisco, Calif, May 1984.

[53] K. Kim and H. Welch. Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Transactions on Computers*, 38(5):626–636, May 1989.

[54] J. C. Knight. Safety critical systems: challenges and directions. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 547–550, New York, NY, USA, 2002. ACM.

[55] S. Konrad and B. H. C. Cheng. Requirements patterns for embedded systems. In *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 127–136, Washington, DC, USA, 2002. IEEE Computer Society.

[56] S. Konrad, B. H. C. Cheng, and L. A. Campbell. Object analysis patterns for embedded systems. *IEEE Transactions on Software Engineering*, 30(12):970–992, 2004.

[57] I. Koren and C. Krishna. *Fault-Tolerant Systems.* Morgan Kaufmann, San Francisco, 2007.

[58] R. Lajoie and R. Keller. *Object-Oriented Technology for Database and Software Systems*, chapter Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts and Motifs in Concert, pages 295–312. World Scientific Publishing, Singapore, 1995.

[59] J. Laprie, J. Arlat, C. Béounes, and K. Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *IEEE Transactions on Computer*, 23(7):39–51, July 1990.

[60] J. Laprie, J. Arlat, C. Béounes, and K. Kanoun. *Architectural Issues in Software Fault-Tolerance, Software Fault Tolerance*, chapter 3, pages 47–80. John Wiley & Sons. Ltd., New York, USA, 1995.

[61] J. Laprie, J. Arlat, C. Béounes, K. Kanoun, and C. Hourtolle. Hardware and software fault tolerance: Definition and analysis of architectural solutions. In *Proceedings of the 17th IEEE International Symposium on Fault-Tolerant Computing*, pages 116–121. IEEE Press, 1987.

[62] J.-C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *25th International Symposium on Fault-Tolerant Computing, 1995, "Highlights from Twenty-Five Years"*, pages 2–11, Jun 1995.

[63] G. Latif-Shabgahi, J. Bass, and S. Bennett. A taxonomy for software voting algorithms used in safety-critical systems. *IEEE Transactions on Reliability*, 53(3):319–328, Sept. 2004.

[64] T.-Y. Lee and P.-A. Hsiung. Embedded software synthesis and prototyping. *Consumer Electronics, IEEE Transactions on*, 50(1):386–392, Feb 2004.

[65] Y.-W. Leung. Maximum likelihood voting for fault-tolerant software with finite output-space. *IEEE Transactions on Reliability*, 44(3):419–427, Sept. 1995.

[66] N. G. Leveson. *Safeware: system safety and computers.* ACM, New York, NY, USA, 1995.

[67] R. Lutz. Software engineering for safety: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 213–226, New York, NY, USA, 2000. ACM.

[68] M. Lyu. *Handbook of Software Reliability Engineering.* McGraw-Hill and IEEE Computer Society Press, New York, 1996.

[69] D. Manolescu, M. Voelter, and J. Noble. *Pattern Languages of Program Design 5.* Addison-Wesley Professional, 2005.

[70] M. Marseguerra and E. Zio. *Basics of the Monte Carlo method with application to system reliability.* LiLoLe–Verlag, Berlin, 2002.

[71] P. Marwedel. *Embedded System Design.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2^{nd} edition, 2006.

[72] D. McAllister, C.-E. Sun, and M. Vouk. Reliability of voting in fault-tolerant software systems for small output-spaces. *IEEE Transactions on Reliability*, 39(5):524–534, Dec. 1990.

[73] MISRA. *Development Guidelines for Vehicle Based Software.* The Motor Industry Software Reliability Association, November 1994.

[74] T. Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers.* Newnes, 2005.

[75] B. Parhami. Voting algorithms. *IEEE Transactions on Reliability*, 43(4):617–629, Dec. 1994.

[76] B. Parhami. Design of reliable software via general combination of N-version programming and acceptance testing. In *Proc. Seventh International Symposium on Software Reliability Engineering*, pages 104–109, 30 Oct.–2 Nov. 1996.

[77] J. K. Peckol. *Embedded Systems: A Contemporary Design Tool.* Wiley Publishing, 2007.

[78] M. Pont. *Patterns for Time-Triggered Embedded Systems: Building reliable applications with the 8051 family of microcontrollers.* ACM Press, New York, 2001.

[79] M. J. Pont and M. P. Banner. Designing embedded systems using patterns: a case study. *Journal of Systems and Software*, 71(3):201–213, 2004.

[80] G. Pucci. On the modelling and testing of recovery block structures. In *Proc. 20th International Symposium Fault-Tolerant Computing FTCS-20*, pages 356–363, 26–28 June 1990.

[81] L. L. Pullum. *Software fault tolerance techniques and implementation.* Artech House, Inc., Norwood, MA, USA, 2001.

[82] B. Randell. System structure for software fault tolerance. *IEEE Transaction in Software Engineering*, 1(2):220–232, 1975.

[83] B. Randell. Dependability, structure and infrastructure. Technical Report CS-TR-877, School of Computing Science, University of Newcastle upon Tyne, November 2004.

[84] B. Randell, J. Horning, H. Lauer, and P. Melliar-Smith. A program structure for error detection and recovery. *LNCS*, 16:171–187, 1974.

[85] B. Randell and J. Xu. *Software Fault Tolerance*, chapter The Evolution of the Recovery Block Concept. John Wiley & Sons, New York, 1995.

[86] F. Redmill. IEC 61508: Principles and use in the management of safety. *IEE Computing and Control Engineering*, 9(10):205–213, 1998.

[87] D. Riehle and H. Züllighoven. A pattern language for tool construction and integration based on the tools and materials metaphor. *Pattern languages of program design*, pages 9–42, 1995.

[88] F. Rincon, F. Moya, J. Barba, and J. C. Lopez. Model reuse through hardware design patterns. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 324–329, Washington, DC, USA, 2005. IEEE Computer Society.

[89] L. Rising. *Design patterns in communications software.* Cambridge University Press, New York, NY, USA, 2001.

[90] S. Robertson and J. Robertson. *Mastering the Requirements Process (2nd Edition).* Addison-Wesley Professional, 2006.

[91] RTCA. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification.* RTCA, Inc., 1992.

[92] RTCA. *DO-254, Design Assurance Guidance for Airborne Electronic Hardware.* RTCA, Inc., 2000.

[93] B. Rubel. Patterns for generating a layered architecture. *Pattern languages of program design*, pages 119–128, 1995.

[94] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns : Integrating Security and Systems Engineering.* John Wiley & Sons, March 2006.

[95] L. Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, July–Aug. 2001.

[96] D. J. Smith and K. G. L. Simpson. *Functional safety: A straightforward guide to applying the IEC 61508 and related standards.* Elsevier, Burlington, U.K., 2nd edition, 2005.

[97] N. Storey. *Safety-Critical Computer System.* Addison-Wesley, Harlow, England, 1996.

[98] N. F. Vaidya and D. K. Pradhan. Fault-tolerant design strategies for high reliability and safety. *IEEE Transactions on Computers*, 42(10):1195–1206, 1993.

[99] W. H. Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, July 1994.

[100] W. Wu and T. Kelly. Safety tactics for software architecture design. In *the 28th Annual International Computer Software and Applications Conference (COMPSAC '04)*, pages 368–375, Washington, DC, USA, 2004. IEEE Computer Society.

[101] L. Xu, H. Ziv, D. Richardson, and T. A. Alspaugh. An architectural pattern for non-functional dependability requirements. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–6, 2005.

[102] N. Yoshida. Design patterns applied to object-oriented SOC design. In *10th Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI 2001)*, Nara, Japan, Oct. 2001.

[103] N. Yoshioka, H. Washizaki, and K. Maruyama. A survey on security patterns. *Progress in Informatics:*, 5:35–47, 2009.

[104] Y. Yu and B. Johnson. The quantitative safety assessment for safety-critical software. In *Proc. 29th Annual IEEE/NASA Software Engineering Workshop*, pages 150–162, 2005.

Curriculum Vitae

Name	Ashraf Armoush
Date of birth	11.11.1977
Place of birth	Awarta, West Bank

2007 – 2010	Research assistant at the Embedded Software Laboratory at RWTH Aachen University, Germany
2003 – 2006	Instructor at An-Najah National University, Palestine
2001 – 2003	Study of master in computer science at the University of Jordan, Jordan
2000 – 2001	Teaching assistant at An-Najah National University, Palestine
1995 – 2000	Study of electrical engineering at An-Najah National University, Palestine
1993 – 1995	Houwarah Boys Secondary School, Palestine
1983 – 1993	Awarta Elementary School, Palestine