STATUS OF THESIS

Title of thesis	Biologically Inspired Self-Healing Software System Architecture

I MAZIN ELHADI

hereby allow my thesis to be placed at the Information Resource Center (IRC) of Universiti Teknologi PETRONAS (UTP) with the following conditions:

1. The thesis becomes the property of UTP.
2. The IRC of UTP may make copies of the thesis for academic purposes only.
3. This thesis is classified as

☐ Confidential

[✓ Non-confidential]

If this thesis is confidential, please state the reason:

____

The contents of the thesis will remain confidential for ______________ years.

Remarks on disclosure:

____

Endorsed by

[illegible]

MAZIN ELHADI

Date: 24/7/08

[illegible]

AZWEEN ABDULLAH
Universiti Teknologi PETRONAS
Bandar Seri Iskandar, Tronoh, 31750
Perak, Malaysia

Date: 24/7/08

UNIVERSITI TEKNOLOGI PETRONAS

Approval by Supervisor (s)

The undersigned certify that they have read, and recommend to The Postgraduate Studies Programme for acceptance, a thesis entitled "**Biologically Inspired Self-Healing Software System Architecture**" submitted by **(Mazin Elhadi)** for the fulfillment of the requirements for the degree of Master of Science in Information Technology.

____

Date

Signature	:	[illegible]
Main Supervisor	:	Dr Azween Bin Abdullah Senior Lecturer Information Technology/Information Systems Universiti Teknologi PETRONAS 31750 Tronoh Perak Darul Ridzuan
Date	:	24/7/08.
Signature	:	
Co-Supervisor	:	
Date	:	

TITLE PAGE

UNIVERSITI TEKNOLOGI PETRONAS

Biologically Inspired Self-Healing Software System Architecture

By

Mazin Elhadi

A THESIS
SUBMITTED TO THE POSTGRADUATE STUDIES PROGRAMME
AS A REQUIREMENT FOR THE
DEGREE OF MASTER OF SCIENCE
INFORMATION TECHNOLOGY
BANDAR SERI ISKANDAR,
PERAK

JUNE, 2008

DECLARATION

I hereby declare that the thesis is based on my original work except for quotations and citations which have been duly acknowledge. I also declare that it has not been previously or concurrently submitted for any other degree at UTP or other institutions.

Signature	:	[illegible]
Name	:	Mazin Elhadi
Date	:	24/7/08

ABSTRACT

Self-healing capabilities have begun to emerge as an interesting and potentially valuable property of software systems. Self-healing characteristic enables software systems to continuously and dynamically monitor, diagnose, and adapt itself after a failures has occur in their components. Adding such characteristic into existing software systems is immensely useful and valuable for allowing them to recover from failures. However, developing such self-healing software systems is a significant challenge.

The nature introduces to us unforeseen concepts in terms of presenting biological systems that have the ability to handle its abnormal conditions. Based on this observation, this thesis presents self-healing architecture for software system based on one of the biological processes that have the ability to heal by itself (the wound-healing process). The self-healing architecture provides software systems the ability to handle anomalous conditions that appear among its components. The presented architecture is divided into to layers, functional and healing layer. In the functional layer, the components of the system provide its services without any disruptions. The component is considered as faulty component if it fails to provide its services. The healing layer aims to heal the faulty component and return it to the running system without the awareness of the user.

The presented self-healing software system is formally described to prove its functionality. Set-theoretic and Finite State Machine (FSM) is introduced. A prototype for the presented architecture has been implemented using Java language. Java objects are considered as the system components. The modules of the healing layer in the self-healing architecture have been implemented into Java classes. An object from the module class will be created to perform its task for the healing process. The thesis concludes with recommendations for future works in this area and enhancement of the presented architecture.

ACKNOWLEDGEMENTS

First of All, I would like to express my gratitude and appreciation to my advisor Dr. Azween B Abdullah. He was always there for me. His support, encouragement, valuable assistance, and unwavering patience rescued me from frustration and guided me throughout my research. Without his excellent guidance it would have been impossible to finish this work. His very positive influence on my professional development will be carried forward into my future career.

I would like to thank my parents, my sister, and my brothers for their endless support. They gave me courage and strength to never give up. They provided me sturdy motivation and strong emotional and moral support which I have needed throughout my life.

Special thanks are extended to my colleagues at Computer and Information Sciences Department. They have been very friendly and gracious which provided me an excellent study environment.

At the end, I would like to thank my friends for their consistent support, help, and encouragement.

TABLE OF CONTENTS

STATUS OF THESIS....................i
TITLE PAGE....................iii
DECLARATION....................iv
ABSTRACT....................v
ACKNOWLEDGEMENTS....................vi
TABLE OF CONTENTS....................vii
LIST OF TABLES....................x
LIST OF FIGURES....................xi
CHAPTER ONE: INTRODUCTION....................1
1.1 Introduction....................1
1.2 Motivation....................3
1.3 Subject....................3
1.4 Objectives....................3
1.5 Scope....................4
1.6 Methodology....................4
1.7 Outline of the Thesis....................5
1.8 Conclusion....................5
CHAPTER TWO: LITERATURE REVIEW....................6
2.1 Introduction....................6
2.2 Autonomic Computing....................6
2.3 Autonomic Computing Classifications....................9
2.3.1 Self-Configuring....................9
2.3.2 Self-Healing....................10
2.3.3 Self-Optimizing....................10
2.3.4 Self-Protecting....................10
2.4 Self-Management Phases....................12

TABLE OF CONTENTS – CONTINUED

2.5 Autonomic Computing Element Architecture..15

2.6 Self-Healing Software Systems..17

2.6.1 Failure Detection..17

2.6.2 Fault Diagnosis..17

2.6.3 Fault Healing..18

2.6.4 Verification..18

2.7 Related Works..20

2.7.1 Autonomic Computing..20

2.7.2 Biological Self-Management..24

2.8 Conclusion..26

CHAPTER THREE: SELF-HEALING SOFTWARE SYSTEM ARCHITECTURE.....27

3.1 Introduction..27

3.2 Wound-Healing..27

3.2.1 Wound-Healing phases..28

3.2.1.1 Homeostasis Phase..29

3.2.1.2 Inflammatory Phase..29

3.2.1.3 Proliferation Phase..29

3.2.1.4 Remodeling or Maturation Phase..30

3.3 Mapping Wound-Healing into Self-Healing Software System........................31

3.3.1 Homeostasis Phase into Fault Control Phase..32

3.3.2 Inflammatory Phase into Repair Phase..32

3.3.3 Proliferation Phase into Repair Validation Phase..32

3.3.4 Remodeling Phase into Integration Phase..33

3.4 Biological Self-Healing Software System Architecture..34

3.4.1 The Proposed Architecture..34

3.4.2 Formal Description..42

3.4.2.1 Definition 1..43

TABLE OF CONTENTS – CONTINUED

3.4.2.2 Definition 2 49
3.4.2.3 Definition 3 49
3.4.2.4 Definition 4 49
3.4.3 Commutativity Property 51
3.4.4 Closure Property 53
3.5 Graphnet Model 56
3.6 Conclusion 73
CHAPTER FOUR: SELF-HEALING SOFTWARE PROTOTYPE 74
4.1 Introduction 74
4.2 Java Language 74
4.3 Architecture Prototype 75
4.4 Example of Faults in Java Objects 80
4.4.1 Exceptions in Java 80
4.5 Application Tasks 80
4.6 Case Study: ATM System 86
4.6.1 ATM System: The Functional Layer 87
4.6.2 ATM System: The Healing Layer 96
4.7 Results and Discussion 98
CHAPTER FIVE: CONCLUSIONS AND FUTURE WORKS 99
5.1 Conclusions 99
5.2 Future Trends 101
PUBLICATIONS 102
REFERENCES 103
APPENDIX A: TEST PROGRAM LISTING 110

LIST OF TABLES

Table 2-1: Autonomic Computing Classifications..11
Table 3-1: The Description of the Phases in Wound-Healing and Self-Healing Software System 33
Table 3-2: Table of Notations...42
Table 3-3: Commutativity and Closure Properties for the Repair Plans..........................55
Table 4-1: Self-Healing Software (Main Classes)...78
Table 4-2: Output Messages from Healing Modules..84

LIST OF FIGURES

Figure 2-1: Self-Management Classifications..9
Figure 2-2: Feedback Loop in Self-Managed Systems 12
Figure 2-3: Structure of an Autonomic Computing 15
Figure 2-4: Information Flow in Self-Healing Systems...18
Figure 2-5: Scope of Self-Healing In Autonomic Pervasive Computing 19
Figure 3-1: Phases of Wound-Healing..28
Figure 3-2: The Time Life of the Wound Healing Phases..30
Figure 3-3: Mapping the Wound-Healing Process to Self-Healing Software System......31
Figure 3-4: Biologically Inspired Self-Healing Software System Architecture............36
Figure 3-5: Example of Fault Expansion..38
Figure 3-6: Blocking SCS and RCS...39
Figure 3-7: Component's States in the Functional Layer...57
Figure 3-8: Fault Detection Graphnet..59
Figure 3-9: Repair Plan Graphnet..61
Figure 3-10: Mutate Plan Graphnet..63
Figure 3-11: Replicate Plan Graphnet..66
Figure 3-12: Mutate-Replicate Plan Graphnet..67
Figure 3-13: Mutate Plan Graphnet for Concurrent-Faults.......................................70
Figure 3-14: Replicate Plan graphnet For Concurrent-Faults...................................71
Figure 3-15: Mutate-Replicate Plan graphnet For Concurrent-Faults........................72
Figure 4-1: UML Class Diagram for Self-Healing Software System.........................77
Figure 4-2: Sequence Diagram for Self-Healing Software System...........................79
Figure 4-3: Accessing File A from the Running System...81
Figure 4-4: Save Data into Data.txt...82
Figure 4-5: Copying Backup File...83
Figure 4-6: The Output from the Healing Modules..85
Figure 4-7: Welcome Screen..87

LIST OF FIGURES-CONTINUED

Figure 4-8: Login Screen..88
Figure 4-9: ATM System Main Menu...89
Figure 4-10: Account Inquiry...90
Figure 4-11: Withdrawal Money...91
Figure 4-12: Withdrawal Report...92
Figure 4-13: Transfer Transaction: Destination Account.................................93
Figure 4-14: Transfer Transaction: Confirm Operation....................................94
Figure 4-15: Exiting From ATM System 95
Figure 4-16: The Output of the Healing Modules in the Healing Layer....................97

CHAPTER ONE: INTRODUCTION

1.1 Introduction

The rapid growth of the computational power over the past decades contributed to introduce large and complex software-based systems with a variety and great number of components. Moreover, information technology systems are continuing to develop with innovations and changes. The new version of the information technology systems are released from existing technologies. Therefore, programmers need to suit their existing applications to the new technologies. To catch this race, software developers have exploited the rapid growth in computational power in every possible way, producing ever more sophisticated software applications and environments, which results in enormous growth in the number and variety of systems and components (Fuad, 2007). As a consequence, software systems become complex and software architecture become less able to anticipate and control interactions among the components of the systems. This leads to complexity of these systems as well as complexity in the environments where these systems reside. This complexity of software systems causes these systems to be difficult to manage and administer (Mohammad & Koen, 2007) (Paul, Alexander & John, 2005). Software-based systems with this complexity must handle such things as resource variability, changing user needs, and system faults at run-time.

Today, information technology organizations encounter growing challenges in the management and maintenance of large scale software systems because these systems must be active and available 24 hours a day, 7 days a week. Existing control methods and tools are inadequate to mange and administer today's and future software systems. Moreover, while the complexity of these systems continue to grow, the need of skilled persons who install, optimize, protect, and maintain these systems becomes more important.

There have been several attempts to reduce the complexity within these systems by introducing better software engineering practices. However these attempts, the complexity of such systems remains the same as more and more new technologies and systems are being incorporated together. This complexity of the software systems and their environment leads to the idea of autonomic computing (Paul, 2001) (Jeffery & David, 2003). This new area of computing aims to provide software systems that have the ability to handle their complexity by themselves. In other words, autonomic computing is a solution which proposes to reallocate many of the management responsibilities from the administrators to the system itself (Mohammad & Mohsen, 2007). The vision of autonomic computing (Jeffery & David, 2003) is to improve the management of the complex information technology systems by introducing self-management systems for configuration, healing, optimization, and protection purposes. From this vision, the major characteristics of autonomic computing systems (Mazeiar & Ladan, 2005) are self-configuration, self-healing, self-optimization, and self-protection. This work focuses on the second characteristic of autonomic systems, self-healing characteristic.

A system is said to be self-healing if it can recover from failures without external intervention. In other words, the system is capable of automatically re-organizing itself to continue operating after part of it has failed. This is obviously closely related to the notion of fault-tolerance in which a system can operate normally despite experiencing failures. There are finer shades of distinguishing semantics when we relate security, fault-tolerance, survivability and self-healing. We use the term self-healing to mean a wider class of systems and degree of re-organization than is usually denoted by the term fault tolerance.

Specification logics are extensively utilized to verify necessary and inherent properties of self-healing systems. These logics can allow notion of good behavior and abnormal behavior to be formally specified and as a result permit precise reasoning about fault tolerance. We intend to look at their application in self-healing systems which can dynamically reconfigure in response to changes in their environment and allow the succinct specification of both self-healing systems and the properties that they must satisfy.

1.2 Motivation

The turn to the nature has brought us many unforeseen great concepts. Biological systems are able to handle many of the challenges that face these systems, with an elegance and efficiency still far beyond current human artifacts. Based on this observation, bio-inspired approaches have been proposed in the past years as a strategy to handle the complexity of such systems. In other words, human efforts to engineer self-healing systems have had limited success, but nature has developed extraordinary mechanisms for robustness and self-healing over billions of years (Fuad, 2007) (Selvin, David & Steven, 2003). This work aims to introduce self-healing software system architecture based on the observation of one of the biological system (the wound-healing process).

1.3 Subject

The subject of this thesis is biologically inspired self-healing software system architecture. The thesis intends to apply the wound-healing process into software system architecture in order to provide software systems the ability to handle system failures without human intervention.

1.4 Objectives

The objective of this research is to develop a method on how to engineer software systems which have similar high stability and efficiency often found in biological systems. This method must be able to:

1. Monitor its own behaviors in order to detect anomalous behavior.

2. Diagnose the anomalous behavior to find the cause of the problem in order to determine the best way to fix this problem.

3. Recover from the problem to the normal execution of the system.

1.5 Scope

The goal of this research is to develop biologically inspired self-healing architecture for software systems. To achieve this, different issues regarding self-healing characteristic must be addressed. Addressing all of these issues and developing solution for them in a single research project is impractical. Therefore, the essential and more important issues are addressed in this work. The remaining issues will be left as future works.

The scope of this research is three fold:

- To study and analyze a biological system that has the ability to heal by itself.

- To devise a self-healing architecture for software systems based on the biological system in the previous step.

- To develop a prototype to validate the proposed architecture.

1.6 Methodology

This thesis is conducted through: (a) a review of the current status and the relevant work in the area of the autonomic computing in general and in the area of the self-healing systems in particular; (b) analyze these works especially in the area of self-healing systems; (c) identify the biological systems that has the ability to heal by itself; (d) study and analyze the biological system in (c); (e) map the biological system process into a self-healing software system; (f) propose self-healing software system architecture based on the mapping of the biological system process into self-healing software system that mentioned above; (g) formally specify the proposed architecture; (h) develop a prototype to validate the proposed architecture; (i) seek avenues for further research.

1.7 Outline of the Thesis

The rest of the thesis is organized as follows: chapter 2 discusses background information required to comprehend this thesis. The chapter introduces an overview of autonomic computing in general and self-healing systems in particular. Moreover, this chapter investigates different works in the area of autonomic computing and self-healing systems.

We introduce our proposed self-healing software system architecture in chapter 3. In this chapter, we present the theoretic as well as the formal descriptions of the architecture.

Chapter 4 presents the prototype of the proposed self-healing software system architecture.

Finally, Chapter 5 concludes this thesis and presents directions for future work.

1.8 Conclusion

The main purpose of this chapter is to provide the reader a brief description of the research topic which will be conducted through this thesis. The motivation and the subject of the study were introduced. The objectives of this research and the methodology were discussed as well.

CHAPTER TWO: LITERATURE REVIEW

2.1 Introduction

This chapter introduces and discusses the necessary information to comprehend this work and provides the background information on autonomic computing in general and the self-healing in particular.

2.2 Autonomic Computing

The emergence of large, complex and pervasive software application and environment is the result of the rapid growth of the computational power in the last decades. Software system is continuing to grow and dominate most of the systems in the industry and academic area. As these systems continuing to develop and become more complex (Eleni & Nancy, 2006), the interactions between the components of these systems become more complicated. Although there have been numerous works that tried to simplify this interaction (Component Based Software Systems Architecture, Object Oriented Programming, etc), the complexity of such systems remains the same as more new software systems are being developed. A failure that occurs in the system's components is the basic challenge facing these systems. This leads to the idea of autonomic computing where the system can fix the failures of its components by itself.

Autonomic computing is an initiative by IBM (Paul, 2001) in 2001. Paul Horn, senior vice president of IBM Research (Daniel & Jeffery, 2007) (Zach & Sam, 2005) addressed the Agenda conference, an annual meeting of the preeminent technological minds held on October 15th 2001, he suggested a solution, which is: "build computer systems that regulate themselves much in the same way our autonomic nervous system regulates and protects our bodies" (IBM Research, 2001).

The autonomic nervous system (ANS) is "the body's master controller that monitors changes inside and outside the body integrates sensory inputs and effects appropriate response" (Manish & Salim, 2005) (George, Vincent, Ian & Chad, 2006). Examples of ANS's operations include, the heart beat as specific rate depending on the body conditions, the body healing from an injury without any human medical intervention, breathing, the body temperature, and many other systems.

The new computing paradigm, autonomic computing, has changed the view of the fundamental definition of the technology age from one of computing, to one defined by data (IBM Research, 2001). After applying the autonomic computing to the design and implementation of computer systems, software, and storage, these systems will have the following fundamental properties from the user point of view:

- Flexibility: The system will be able to examine data via a platform- and device-agnostic approach.

- Accessible: The nature of the autonomic system is that it is always on.

- Transparent: The system will perform its tasks and adapt to a user's needs without dragging the user into the intricacies of its workings.

The ultimate goal of autonomic computing is to create computer systems that possess the capability and the property of self-management to overcome their growing and handle any failure or fix abnormal situation during their execution. The autonomic computing paradigm transforms computer systems to mature systems. IBM suggests eight characteristics of an autonomic system (IBM Research, 2001):

- "know itself": the autonomic systems needs to know all the information about its components such as: the function of the component, current status, the interaction between components. This information helps the system to manage its self.

- Autonomic configuration: autonomic system must configure itself according to its condition. The configuration is needed to handle the changes in the system environment.

- Autonomic system has the ability to optimize itself by enhancing the interaction between its components.

- Autonomic system must have the ability to recover form components failures. The ability to handle component malfunction called "healing".

- Computer systems are vulnerable to various types of attacks. Therefore, autonomic computing must have the ability to protect its self to maintain overall system security and integrity.

- Autonomic system must interact with other system and use remote resources. As a consequence, autonomic system must have the ability to adapt itself and communicate with the surrounding environment.

- Autonomic system must be able to manage itself as well as functioning in a heterogeneous environment and implement open standards.

- Without involving the user into details, autonomic system must anticipate the optimized resources needed while keeping its complexity hidden.

2.3 Autonomic Computing Classifications

Presenting the autonomic computing concept, IBM established regroup four different classifications of self-management system (Davide, 2004) (Aaron & Charlie, 2005) (Zach & Sam, 2005) (George, Vincent, Ian & Chad, 2006): self-configuring, self-healing, self-optimizing, self-protecting. Figure1-1 illustrates the four self-management classification.

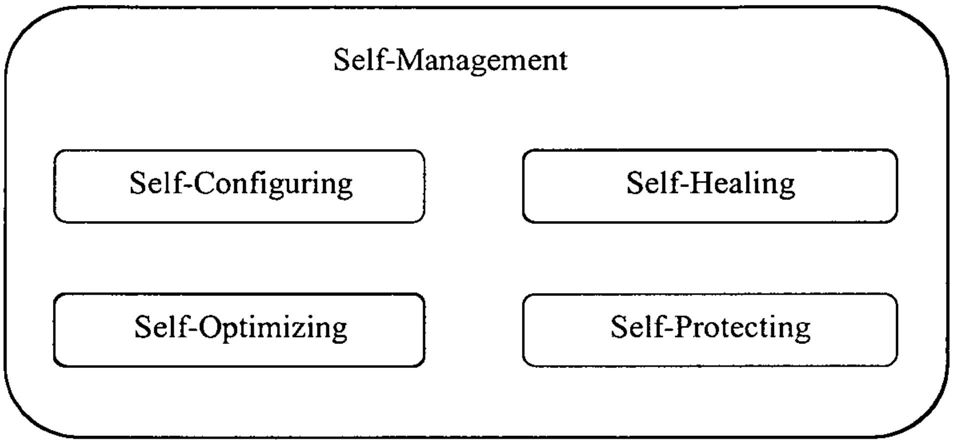


Figure 2-1: Self-Management Classifications

The terminology and means of these classifications are as follows:

2.3.1 Self-Configuring

This classification concentrates on the autonomic system itself. Self-configuring is a mechanism to change the interaction between the system's components in order to improve the services. In other words, it denotes the system ability to change the system structure dynamically. For example, self-configuring systems able to insert new component to the system, replace an existing component with another component, remove component, and control the interaction between these components.

2.3.2 Self-Healing

The system ability to examine, detect any faults or run time anomalies, diagnose, and recover from this fault or runtime anomalies and continue providing its full service easily. The main objective of self-healing is to maximize availability, survivability, maintainability and reliability of the system (Alan & Thomas, 2003). Self-healing system must be able to observe the functionality of its components in order to detect component failure by evaluating the current constraints of the component and apply the appropriate corrections. In other words, self-healing system must have knowledge about its components behavior in order to find, diagnose and recover from the component failure.

2.3.3 Self-Optimizing

An autonomic system must be able to detect any sub-optimal behaviors and optimize itself to improve its execution (Manish & Salim, 2005). Self-optimization is the ability of the system to autonomously optimize its resources. For example, monitoring, optimizing, and allocating the resources in a proper way in order to provide the services.

2.3.4 Self-Protecting

Security is the critical issue in software system. Self-protecting focuses on the system security aspects. Software systems are vulnerable to many types of attack such as non authorized access, viruses, denial-of-service, etc. Therefore, autonomous systems must be able to observe external attacks and as a consequence they should take a specific action in order to make the systems safe, non-vulnerable, and more secure.

Along with these classifications, autonomic system has the following properties (Manish & Salim, 2005) (Mazeiar & Ladan, 2005), table 2-1 shows the four classifications as well as the additional properties:

- Self-Awareness: awareness of its current state and behavior in order to cooperate with other autonomic systems (Deepak, 2005).

- Context-Awareness: awareness of its environment and the ability to react to any changes in its environment. This property some times called self-adaptive which is "software that evaluates and changes its own behavior when the evaluation indicates that it has not accomplishing what it is intended to do, or when better functionality or performance is possible" (Laddaga, 1999).

- Anticipatory: expectation of any changes that may occur to the system state and the ability to manage these changes.

- Openness: ability to integrate with heterogeneous environment and operate with open standard and protocols.

Table 2-1: Autonomic Computing Classifications

Classification	Main Focus
Self-Configuring	Usability
Self-Healing	Dependability
Self-Optimizing	Maintainability
Self-Protecting	Security
Self-Awareness	Functionality
Context-Awareness	Adaptability
Anticipatory	Efficiency
Open	Portability and Integrity

2.4 Self-Management Phases

The capability of a digital system to automatically and dynamically change and adapt its own behavior and characteristic, to improve functionality and dependability is called self-management (Jochen, 2007). Self-management is general concepts of self-CHOP which an abbreviation for; self-configuring, self-healing, self-optimizing, and self-protecting. In some works these classifications are called self-abilities. Next section introduces these classifications in details.

Numerous of systems with self-management capability have been investigated in last decades to address different problems and they regroup all the defined classifications in order to provide a system always efficient without user intervention.

Depending on goals and application domains, self-management mechanisms exploit different approaches to execute a set of common steps. From the classification of self-management systems, the common steps that might be executed by self-management mechanisms are shown in Figure 2-2.

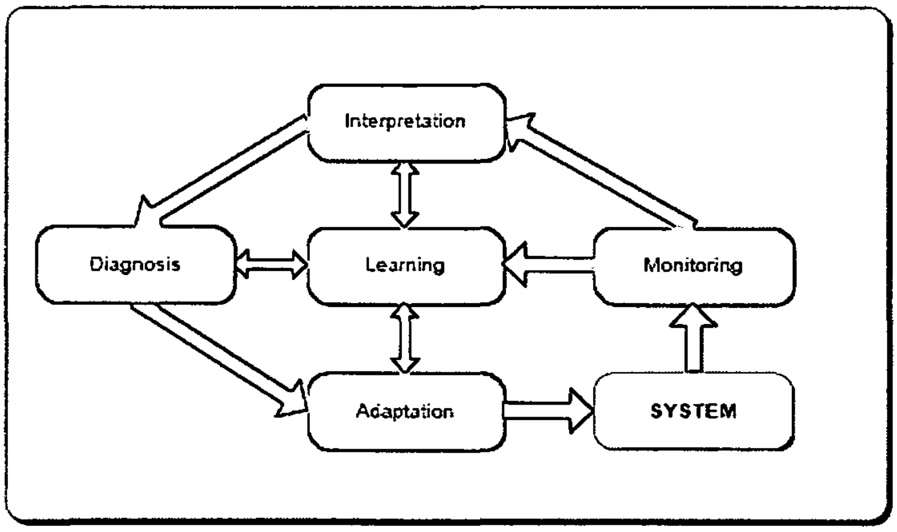


Figure 2-2: Feedback Loop in Self-Managed Systems

Some authors divide the feed back loops in self-managed systems to five common steps; monitoring, interpretation, diagnosis, adaptation, and learning. Other authors divide the feed back loops into more than five steps by splitting diagnosis and adaptation steps into different steps (Grishikashvili. Pereira & Pereira, 2007). Next are the descriptions of each the steps:

- Monitoring: If there is any abnormal condition of the current system behavior is discovered by this step. The rules and conditions are previously defined to support decision making at each step. These steps provide statistical analysis related to the system performances such as CPU usage, memory usage process on execution or network latency. Because this step depends on dynamic data, these data must be compared to the standard data in order to determine if the actual system behavior is not consistent with normal behavior. This step must catch the exception raised by system modules. Moreover, it must provide analysis related to environment where the system is running. Also, the monitoring step must observe the internal behavior of the system as well as the behavior of the operating environment.

- Interpretation: The data which have been collected by the monitoring step is analyzed and verified by the interpretation step. If any abnormal condition has been observed, the detection module tries to retrieve the problem resolution record. If the record for the specified problem is not found, the detection module updates knowledge of the knowledge base module adding the report through the learning module.

- Diagnosis: This step identifies the problem by determining the causes of the problem and verifies the applied solutions.

- Adaptation: Tries to execute the problem resolution cycle. The cycle starts from the solution record identified by the detection module. To complete task, mechanisms are required to dynamically plan, deploy and enact changes, to remove either the diagnosed faults or their effects.

- Learning: Creates and updates the knowledge base acquiring new knowledge learned from data collected by the monitoring activity.

2.5 Autonomic Computing Element Architecture

In any autonomic computing system, autonomic computing elements are the basic building blocks and their interactions produce self-management behavior (Mohammad & Mohsen, 2007). Figure 2-3 illustrates the generic architecture of the autonomic element.

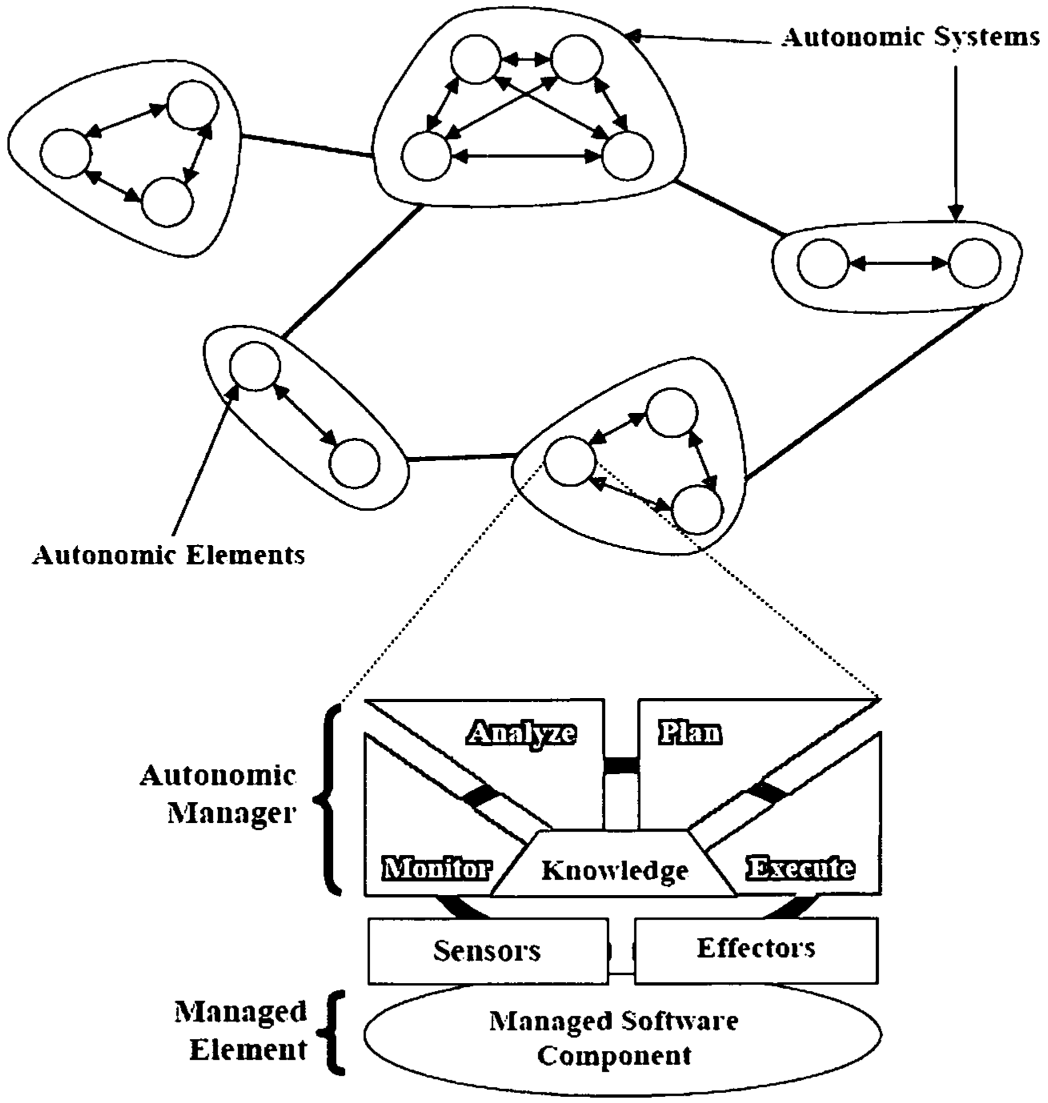


Figure 2-3: Structure of an Autonomic Computing

As shown in Figure 2-3, autonomic computing system consists of autonomic elements. The autonomic elements can communicate with each other and as a consequence autonomic systems can interact with each other. Each autonomic element has self-management property. In other words, each autonomic element knows its own behavior, state, and the interaction with the other elements whether these elements are in the same environment or in other neighboring environments.

The developer sets the goal of the autonomic element to control its behavior and states as well as its interaction with the other autonomic elements. Sometimes, autonomic element helps other autonomic elements to achieve their goals.

In Figure 2-3, each autonomic element has autonomic manager and managed element. The autonomic manager is responsible for the self-management function. It consists of four components:

- Monitor: monitors the autonomic element and itself.

- Analyzer: analyzes the current state of the autonomic element from the data which received from the monitor component.

- Plan Component: based on some constraints and policies, the plan component decides to take the appropriate plan to meet the behavior or state changes.

- Executor: this component executes the plan which devised by the plan component.

The managed element is the component from the system. Sensors and effectors are the basic components for the autonomic element to deal with its environment. Sensors monitor the environment and effectors deliver the control information to the managed element.

2.6 Self-Healing Software Systems

In order to invent self-healing software system architecture, a clear vision of self-healing systems is needed. This section introduces general concepts about self-healing software systems.

The topic of self-healing systems has been studied in a number of areas, including robotics and control systems, programming language design, fault-tolerant computing, and middleware infrastructures (Fabio, Fabio, Gordon & Roy, 2002) (Marija, Nikunj & Nenad, 2002). Self-healing software systems rely on four main phases in order to react to adverse conditions in their runtime environment: failure detection, fault diagnoses, fault healing, and verification of healing actions (Jochen, 2007). Next, we provide a concise description of each phase. Moreover, in each phase we present some works that related to the phase.

2.6.1 Failure Detection

Failure detection denotes to mechanisms that present conditions violate correctness assumptions about the runtime states of the program, usually stated by constraints. This phase is the initial phase of the self-healing process. Therefore, without knowing what has gone wrong in the system, the healing process might not be achieved.

2.6.2 Fault Diagnosis

This phase denotes to mechanisms that analyze the detected failures to the parts of the system that are responsible for these failures.

2.6.3 Fault Healing

The system decides which changes should be applied to the system to fix the detected problems.

2.6.4 Verification

Verification of healing actions makes sure that the conducted measures to overcome the failures do not cause additional problems. The verification phase is often implicit.

Figure 2-4 depicts the general self-healing software systems architecture. The system model contains useful information for each phase.

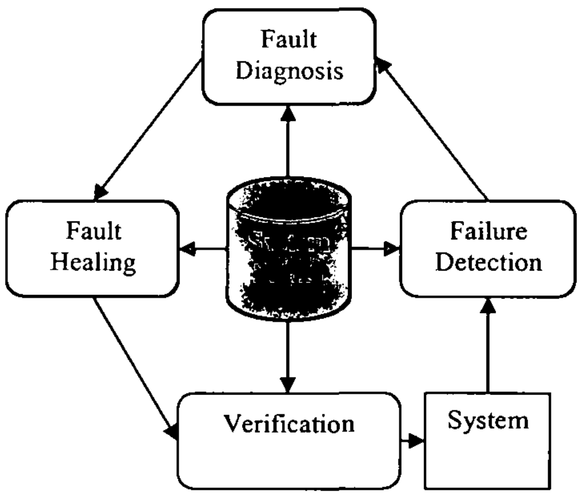


Figure 2-4: Information Flow in Self-Healing Systems.

(Shameem, Sheikh, Moushumi & Munirul, 2007) define the concepts of autonomic computing and self-healing systems. Systems that have the ability to manage itself and dynamically adapt to change in accordance with policies and objectives are termed as autonomic computing. If the system is an autonomous system, it has the ability to identify and correct problems often before they are noticed by the user.

They define self-healing as, self-healing systems that have the ability to perceive those are not operating correctly and, without human intervention, make the necessary adjustment to restore them to normal operation. They determine the scope of self-healing in Figure 2-5.

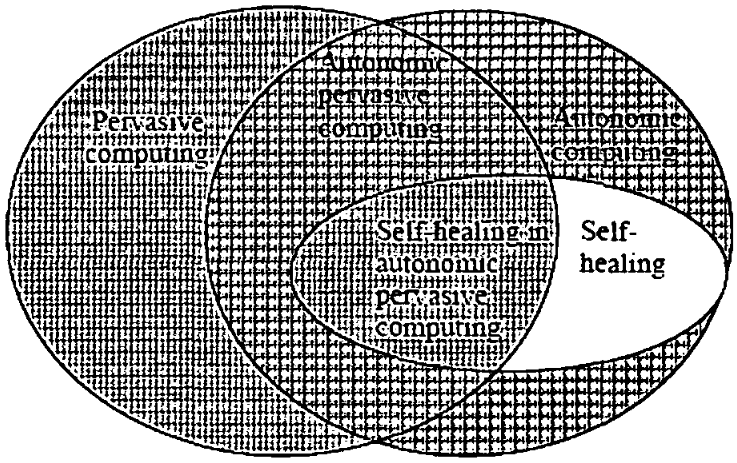

Pervasive computing
Autonomic pervasive computing
Autonomic computing
Self-healing in autonomic pervasive computing
Self-healing

Figure 2-5: Scope of Self-Healing in Autonomic Pervasive Computing

2.7 Related Works

This section presents autonomic computing and self-healing researches in general and biological self-healing researches in particular.

2.7.1 Autonomic Computing

(Dashofy, André & Richard, 2002) create an approach for self-healing systems based on software architecture. The repairs are achieved at the level of a software system's components and connectors. They believe that before an automated planning agent can decide how to repair a self-healing system, a significant infrastructure must be in place to support making the planned repair. Moreover, the self-healing system must be built using a framework that allows for run-time adaptation. Therefore, they present tools and methods that implement these infrastructure elements in the context of an overall architecture-based vision for building self-healing systems. These tools and methods help to express the repair plan, in order to help the reconfiguration agent to execute the repair plan after it is created.

(Michael, 2004) described an approach to designing self-healing components for robust, concurrent and distributed software architecture. A self-healing component is able to detect object anomalies inside of the component, reconfigure inter-component and intra component before and after repairing the sick object. Each component is structured to the layered architecture with two layers, the service layer and the healing layer. The service layer and of a self-healing component provides functional services to other components, whereas the healing layer encapsulates the self-healing mechanism for monitoring objects in the service layer and repairing the sick objects detected. The process of component self-healing includes detection, reconfiguration before and after repairing, repairing, and testing.

(Michael & Daniel, 2005) described the self-healing mechanism for components in reliable systems. Each component in a self-healing system is designed as a layered architecture, structured with the healing layer and the service layer. The healing layer of a self-healing component is responsible for detection of anomalous objects in the service layer, reconfiguration of the service layer, and repair of anomalous object detected. The service layer of a self-healing component provides functionality to other components, which consists of tasks, connectors, and passive objects accessed by tasks. A connector supports the self-healing mechanism for self-healing components as well as encapsulates the synchronization mechanism for message communication between tasks in a component.

(Jochen, 2007) discussed general requirement for failure detection in self-healing software, and proposed an approach to automatically map system level specifications to run-time checkable code-level assertions. This work proposes as automatic technique that addresses the problems of incompleteness and ambiguity by mapping high-level requirements to executable assertions. The technique follows four basic steps: (1) extracting useful information from the requirements. This task is a human task. The software engineer has to identify useful information that implies constraints in the model, (2) Annotating the conceptual model which turns the extracted information from a human readable requirements specification into a format that can be handled automatically. (3) Based on the requirements, which abstract constraints are associated with the entities in the conceptual model, the implied invariants are mapped to operations and relationships pertaining to those entities, this step called mapping constraints to code-level entities. (4) After the determination of which invariants have to be checked, and in which code locations the checks need to be added, then the additional code will be generated and inserted into the program.

(Yang, Yang & Xu, 2005) proposed a framework for the self-healing systems based on dynamic software architectures. Their primary idea is to use the approach to guide the repair of the running system, while the software architecture itself is changeable and manageable during runtime. They divided their framework into two main parts. One part is related to the architecture, consisting of an architecture manager (AMR) and an architecture model container (AMC). The AMR is responsible for monitoring and management of the architectural model. The other part is related to running system, consisting of a running system (RS) and a runtime environment (RE). The RE is responsible for monitoring and reconfiguration of the RS.

(Shin & Jung, 2006) describe an approach of self-reconfiguration. This approach is part of a self-healing mechanism against anomalous objects. The self-reconfiguration is prior to repairing anomalies of objects. The system is structured into components and connectors between the components. The component is self-reconfigured differently in accordance with the object types, such as tasks (concurrent or active objects), connectors between tasks, and passive objects accessed by tasks in the component, while a connector between the components is self-reconfigured in response to the different object types constituting a connector. An asynchronous message queue connector between components is used to illustrate self-reconfiguration of a connector between components.

A modeling framework for self-healing software systems, which proposed by (Michael, Jing, David & John, 2007), is a generic modeling framework to facilitate the development of self-healing software systems. They use a model-based approach to organize software failures and specify their disposition at the model level. The self-healing part is achieved by transforming the model of the system into platform-specific implementation instrumented with failure detection and resolution mechanisms to mitigate the effect of software failures and maintain the level of healthiness of the system.

Failures in software systems are very critical issues in computing. Finding ways to dynamically validate software systems to avoid the high cost of system failures are becoming more imperative. Although research continues to advance in many areas of autonomic computing, there is a lack of development in the area of testing these types of systems at runtime. (King, Babich, Alava, Clarke & Stevens, 2007) propose a framework that dynamically validates changes in autonomic computing systems. They extend the current structure of autonomic computing systems to include self-testing as an implicit characteristic.

A conceptual architecture for fault diagnosis and self healing of interpreted object oriented application has been presented by (Haydarlou, Overeinder & Brazier, 2005). The architecture deals with current and legacy interpreted object oriented code. Their architecture presents a technique which makes the application able to heal itself from failures. Furthermore, the application can attempt to solve the root cause that initiate the fault.

The emergence of the web services has introduced heterogeneous computing systems. These systems can interact dynamically with each other to deliver specific services. (Zeid & Gurguis, 2005) combine the goals of autonomic computing and the promises of web services into one technology called *Autonomic Web Services*. This work aims to provide web services that possess autonomic computing features. The developed technology merges autonomic computing that provides the primitive for achieving self-management, services-oriented architecture that provides the required infrastructure for achieving just-in-time integration among computing systems, and web standards needed for achieving interoperability.

Many other works have been investigated in the area autonomic computing to introduce self-management system (Alessandra, 2007) (Stuart et al., 2003) (Bogdan, Dan, Marin & Mircea, 2007). These works have not adopted the characteristics that found in biological systems. In other words, these works have addressed the requirements of autonomic computing from software engineering point of view.

2.7.2 Biological Self-Management

(Wang, Li & Bu, 2004) proposed a biological formal architecture of self-healing system. They presented a self-healing system model for unstable network environment. The introduced model treats a self-healing system as a component-based architectural model which comprise business logic module and control module. The model introduces a definition of heart, sensor and DNA-logic. Heart is the system kernel module represented as three element tuple (Curlnfo, DNAlogic, Cond), Curlnfo is the current information of the component, DNAlogic is the evolution result of the component, Cond is the condition to be met if the component evolution shall occur. Sensor is a finite set of sensors collecting the component's run-time information and sending it to the heart module.

A biologically-inspired autonomic architecture for self-healing data centers, called SymbioticSphere, was proposed by (Paskorn & Junichi, 2006). This architecture allows data centers to autonomously adapt to dynamic environmental changes and survive partial system failures. The architecture follows certain biological principles such as decentralization, natural selection, emergence and symbiosis to design data centers (application services and middleware platforms). Each service and platform is modeled as a biological entity, analogous to an individual bee in a bee colony, and implements biological concepts such as energy level, health level, energy exchange, environment sensing, migration, replication and death. Simulation results show that, like in biological systems, desirable system properties in data centers (e.g., adaptability and survivability) emerge from collective actions and interactions of services and platforms.

(Selvin, David & Lance, 2002) propose a cell-based programming model that can be used for software systems operation and healing. The model is more closely related to the biological processes. Their model supports a notion of cell division, a communication model based on chemical diffusion, and a rudimentary model of the physical forces involved. They represent a cell program as an automaton containing discrete states and transitions between the states. Every cell comprising the program is in one of these states.

The input to each cell state is the sensed properties of the local environment and the output is a transition to another state, or a division into two (possibly different) states. They also applied the concept of the Nature's programs which are encoded in DNA and exhibit remarkable density and expressiveness.

(Pruet & Junichi, 2002) propose middleware architecture for sensor networks (A biologically-inspired middleware architecture for self-managing wireless sensor networks, BiSNET). This work addresses several key issues in multi-modal wireless sensor networks (MWSNs) such as autonomy, scalability, adaptability, self-healing and simplicity. The difference which makes this work attractive is that this work is based on the observation that various biological systems have developed mechanisms to overcome these issues. BiSNET follows certain biological principles such as decentralization, food gathering/storage and natural selection to design MWSN application. They present some biological systems such as bees and their interaction with each other and food gathering and storage.

Biologically inspired self-governance and self-organisation for autonomic networks is techniques which proposed by (Sasitharan, Dmitri, William, Mícheál & John, 2006). As the autonomic network management provides the ability for network devices to cooperatively self-organise and self-govern in the support of high level business goals, they argue that these principles are inspired by biological systems. They propose key self-organisation and self-governance techniques that are drawn from principles of molecular biology. The biological processes that included in their works are blood glucose homeostasis, reaction diffusion, microorganism mobility using chemotaxis techniques, and hormone signaling.

Although many biological autonomic computing approaches have been proposed, none has fully adopted and implemented a complete biological process. These works have adopted small parts of the biological process such as DNA, cell division, and chemical diffusion.

2.8 Conclusion

This chapter has introduced the fundamental information that helps the reader to understand the basic concepts and terms presented in the rest of this thesis. Sections 2.2 until 2.5 have introduced the term autonomic computing and its classifications. Section 2.6 has presented some related works. The different between these works and this thesis is that this thesis adopts a complete biological process modeling to introduce the self-healing software system architecture.

CHAPTER THREE: SELF-HEALING SOFTWARE SYSTEM ARCHITECTURE

3.1 Introduction

Software architectures provide high-level abstractions for representing the structure, behavior, and key properties of a software system (Dewayne & Alexander, 1992). These abstractions involve descriptions of the elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on those patterns (Mary & David, 1996).

This chapter introduces software system architecture based on the wound-healing processes. We believe that, to devise biologically self-healing software system architecture one needs to observe one of the self-healing processes of the biological systems. We found that, one of the biological systems that have the ability to heal by themselves is the wound-healing process.

In this chapter, we introduce the definition of wound-healing as well as the description of its phases. Then, we explain how we mapped each phase of the wound-healing process to the expected phase in self-healing software system. Finally, we present our proposed architecture based on the mapping.

3.2 Wound-Healing

One of the biological systems that have the ability to heal by themselves is the wound-healing. In this section, we intend to present brief, clear and simple description of wound-healing process.

A wound is created when the anatomic integrity of the tissue is disrupted and healing is the process whereby the integrity of the tissue is restored ("WoundHeal"). The biological process that starts with an injury (the disruption of the anatomic integrity of the tissue) and ends with formation (restoring the integrity of the tissue) is called wound-healing (or wound repair).

The wound-healing process is complex, dynamic, and continuous process. It consists of distinct phases which overlap in time. Some authors categorize the wound healing process into three separate phases (the inflammatory, proliferation, and remodeling), and the others categorize it to four or more by dividing the inflammatory or proliferation phase into different phases (David & Heather) (Jayne & Sarah, 2004) (Sarah, 2002).

3.2.1 Wound-Healing Phases

In this section, we introduce the wound-healing phases. According to (David & Heather) (Robert & Melissa, 2004), the wound-healing process consists of four phases. Figure 3-1 illustrates the sequence of wound healing process.

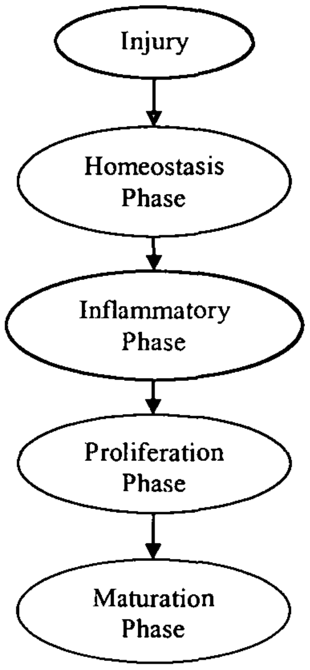


Figure 3-1: Phases of the Wound-Healing Process

3.2.1.1 Homeostasis Phase

After the disruption of the anatomic integrity of the tissue occurs, the body responds quickly to this disruption. Within seconds after the injury, the blood vessels constrict to stop bleeding at the site. Platelets, cells that produce substances to aid in stop blood bleeding; prime role is to form a stable clot sealing the damaged vessel. To stop blood bleeding, platelets aggregate and adhere to exposed collagen-protein of connective tissue-to initiate the second phase.

3.2.1.2 Inflammatory Phase

The second phase of wound-healing presents as swelling and warmth often associated with pain. The basic work to be done in wound-healing is to clean up the debris. The blood vessels become leaky and releasing plasma into the surrounding tissue because of the inflammatory response.

In this phase, there are some types of cells which act as the first line of defense against infection called *neutrophils*. Another type of cells which acts as the second line of the defense called *macrophages*. These cells task is starting rebuilding (or repairing) the injury site. Also, they produce variety of substances which appear to direct the next phase.

3.2.1.3 Proliferation Phase

In this phase, epidermal cells burst into mitotic activity. Then, they begin their migration across the surface of the wound. Another type of cells, fibroblasts, proliferates in the deeper part of the wound. This type of cells begins to produce small amount of collagen and *proteoglycans* which acts as a scaffold for migration and further fibroblast migration.

At the end of this phase the fibroblasts begin to produce large amount of collagen and proteoglycans. Collagen fibers are laid down randomly and cross liked into large, closely and packed bundle.

3.2.1.4 Remodeling or Maturation Phase

The remodeling in the wound-healing process involves remodeling of the dermal tissue to produce greater tensile strength. After the fibroblasts leave the wound site, collagen is remodeled into a more organized matrix.

As we mentioned earlier in this section, the wound-healing process overlap in time. The homeostasis phase begins immediately after the injury and the next phase begins within seconds. The second phase, inflammatory phase, takes 1-4 days. Proliferation phase starts four days after wounding and usually lasts until day 21. The last phase lasts up to years after wounding. Figure 3-2 shows the time life of the wound healing process.

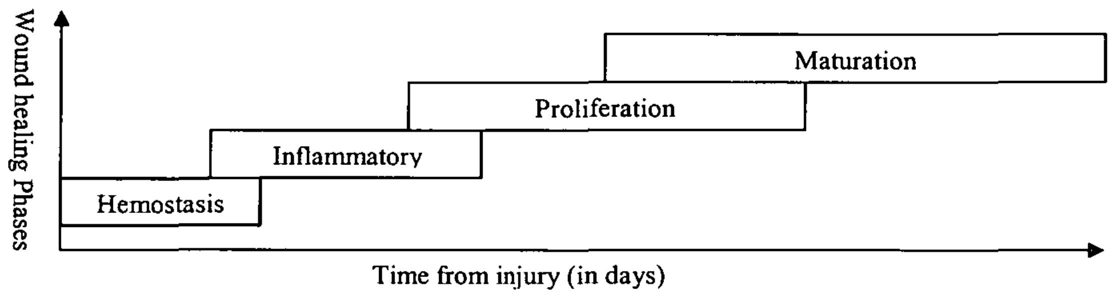


Figure 3-2: The Time Life of the Wound-Healing Phases

Infection of bacteria, fungus, or virus which interrupts the healing process must be controlled during the wound-healing phases in order to perform the normal healing process.

3.3 Mapping Wound-Healing into Self-Healing Software System

In this section we show how the wound-healing process can be mapped into self healing software system. We mapped each phase in the wound-healing process to the expected phase in self-healing software system (see Figure 3-3).

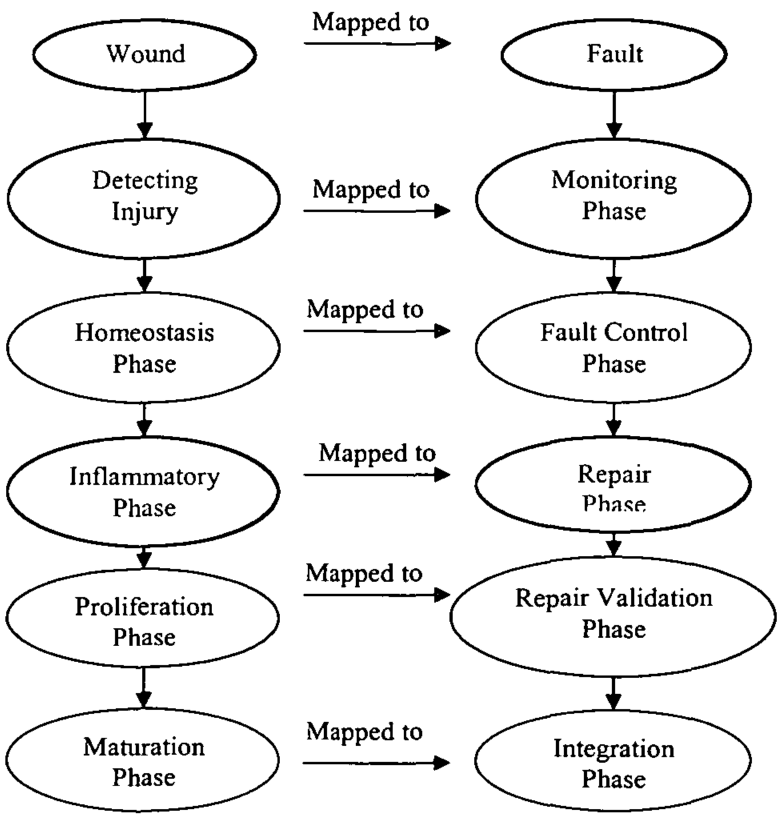


Figure 3-3: Mapping the Wound-Healing Process into Self-Healing Software System

In the wound-healing, a wound is created when the anatomic integrity of the tissue is disrupted. In software systems, a fault is a structural imperfection in a software system that may lead to the system's eventually failing. Therefore, we mapped the wound to the fault.

In the wound-healing process, when an injury happens, the body sends particular chemical signals to indicate that there is an injury at the specified area. In software systems, when a fault occurs in one of the system's components, a particular technique is needed to detect this failure.

3.3.1 Homeostasis Phase into Fault Control Phase

In wound-healing process, the body tries to stop bleeding at the injury site. So, too in software systems, the system needs to stop losing components that are related to the faulty component. In wound-healing, special cells are sent to the injury site either to stop blood bleeding or to produce substances that help in stopping the blood bleeding. To map this phase to self-healing software system, we need to find a technique that stop losing components.

3.3.2 Inflammatory Phase into Repair Phase

In this phase, in order to start the healing process after detecting the injury and stopping the blood bleeding, the body starts to clean up the debris at the injury site. Also, special types of cells start to produce some substances which help on rebuilding (repairing) the injury site. In software system, isolating the faulty component from other components is needed. This isolation prevents the other components from failure. Moreover, repairing (healing) the faulty component becomes easier when the faulty component is isolated from other component.

3.3.3 Proliferation Phase into Repair Validation Phase

Two types of cells appear in this phase. The first type of cells migrates across the surface of the wound. The second type of cells proliferates in the deeper part of the wound. These two types of cells produce large amounts of substances which contribute in creating and connecting new tissues and blood vessels. By the end of this phase, new tissues and blood vessels are produced. In software system, repairing the faulty component results a new component, the healed component. Therefore, testing the new component (the healed component) is needed to make sure that the healed component is working in a proper way.

3.3.4 Remodeling Phase into Integration Phase

The role of the last phase in wound-healing process is to remodel the tissues and strengthen the scar. Also, collagen fibers are remodeled to more organized matrix. So too in software system, we need to return the healed component to the running system without affecting the other components.

Table 3-1 summarizes the task of each phase in the wound-healing process as well as in the self-healing software systems.

Table 3-1: The Description of the Phases in Wound-Healing and Self-Healing Software System

Wound-Healing		Self-Healing Software System	
Phase	Description	Phase	Description
Hemostasis	Stop bleeding	Fault Control	Stop losing other components
Inflammatory	Removing the debris at the injury site	Repair	Isolating and repairing the faulty component
Proliferation	Build and fill the injury site	Repair Validation	Test the healed component
Remodeling/Maturation	Remodeling the tissues and Strengthen the scar	Integration	Returning the healed component to the system

We believe that, infection of viruses should be controlled in the wound-healing process as well as in self-healing software systems during the healing time. As we mentioned in chapter two, there is another area of autonomic computing that focuses on security issues called self-protecting.

3.4 Biological Self-Healing Software System Architecture

In the previous sections we discussed how we mapped the phases of the wound-healing process into phases of the self-healing software system. In this section, we introduce our self-healing software system architecture.

In wound-healing process, particular types of cells are responsible for particular tasks. For example, in the first phase of wound-healing process, the homeostasis phase, blood vessels constrict to stop bleeding and platelets adhere and produce special substances which help to stop bleeding. Likewise, in our self-healing software system architecture, we introduce some modules in each phase. These modules play the same role of the cells in the wound-healing process. In other words, each module is responsible for a specific task. Figure 3-4 depicts the phases of our self-healing software architecture as well as there modules.

3.4.1 The Proposed Architecture

Our self-healing software system architecture consists of two layers: the functional layer and the healing layer.

1. **The Functional layer**

   In this layer, the system executes normally without any fault. In other words, the system provides its full services in this layer. Each component in the system provides its full service and interacts with other components without any disruption. For example, in Figure 3-4, the system consists of four components C1, C2, C3, and C4.

2. The Healing Layer

If one of the components fails to provide its services during the execution of the system (receive input, process, deliver output), the component is considered as faulty component. In this layer we aim to return the faulty component to its normal condition (the functional layer) by applying the wound healing phases in self-healing software systems. The healing layer composes of five phases: Monitoring phase, Fault Control Phase, Repair Phase, Repair Validation Phase and Integration Phase. Each phase consists of a set of modules. These modules interact with each other to achieve the task of their phase. The modules are numbered from 1 to 10 in Figure 3-4.

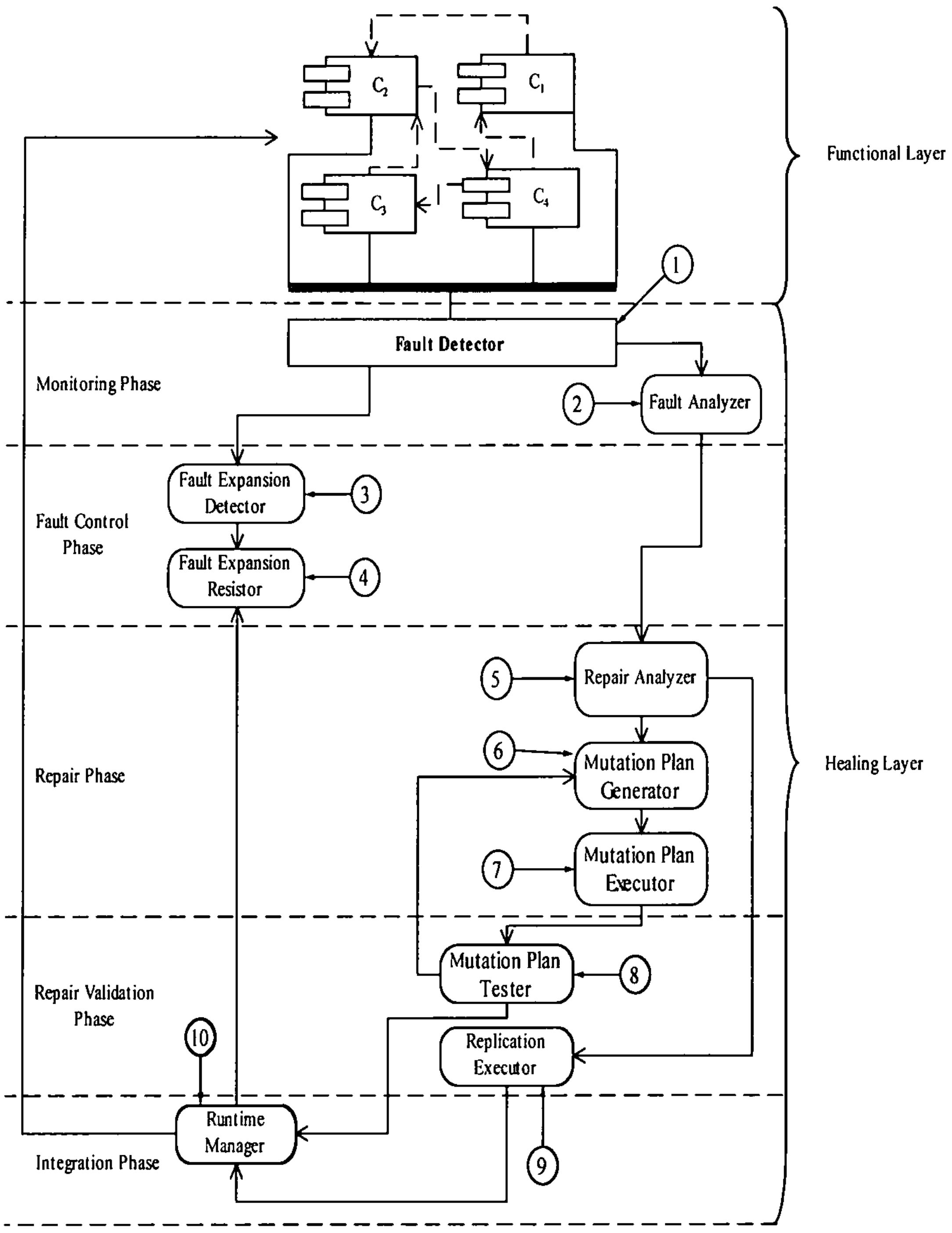


Figure 3-4: Biologically Inspired Self-Healing System Architecture

- **Monitoring Phase (Failure Detection Phase)**

This phase consists of two elements; Fault Detector and Fault Analyzer. These two elements act like the chemical signals in wound-healing process which would be sent once the injury is discovered.

- Module No.1: Fault Detector

The task of this element is to observe the component's behavior by monitoring its execution. The normal execution of the component is expressed by constraints. If a fault occurs during the component's runtime execution (revealing conditions that violate correctness assumption about the execution of the component), the Fault Detector sends two messages; the first message containing the fault information (for example fault time and the current conditions of the component) will be sent to the Fault Analyzer, the second message will be sent to the Fault Expansion Detector (in the Fault Control Phase). This message notifies the Fault Expansion Detector that the component has failed.

- Module No.2: Fault Analyzer

This module analyzes the cause of the fault (for example; determining the root that causes the fault and whether the cause of the fault internal or external).

- **Fault Control Phase**

The task of the homeostasis phase in wound-healing process is to stop blood bleeding after the injury is detected. In this phase we aim to stop the expansion of the fault. If one of the components of the system fails, this fault may affect the other components that are related to the faulty component. Figure 3-5 shows an example.

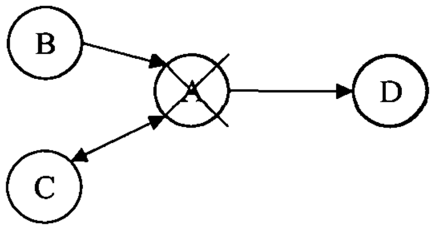


Figure 3-5: Example of Fault Expansion

In the normal execution, component A receives messages from component B and C, and sends two messages; one to component D and the other back to component C.

If component A fails to provide its service, which means component A will not be able to receive messages from other components and each output message will be a wrong output which might affect the other components by sending the wrong data or messages. This also might lead to failures in components C and D. In this case, we need to stop component A from sending and receiving messages, in other, words we need to isolate component A.

In the wound-healing process, two types of cells are responsible for stopping the bleeding; platelets and blood vessels. To achieve this in our model, we provide two elements:

• Module No. 3: Fault Expansion Detector

The task of this element is to create two sets: the first set, called Sender Components Set (SCS), contains the components that send messages to the faulty component (for example, in Figure 3-5, if the faulty component is A, SCS set contains B and C,) the second set, called Receiver Components Set (RCS) contains the components that receive messages from the faulty component (for example, in Figure 3-5, if the faulty component is A, RCS set contains C and D.) In wound-healing process, Platelets produce special substances that aid to stop the bleeding. Likewise, Fault Expansion Detector generates the two sets (SCS, RCS), which aid in stopping fault expansion, and sends these sets to the Fault Expansion Resistor.

• Module No.4: Fault Expansion Resistor:

After receiving the two sets, this module blocks the components that related to the faulty component from sending/receiving messages to/from the faulty component. (for example, in Figure 3-6 component B, C and D will be blocked from sending/receiving messages to/from the faulty component A). To stop bleeding, the blood vessels constrict in the area of the wound area. In the same way, Fault Expansion Resistor is responsible for stopping the fault from spreading to other components.

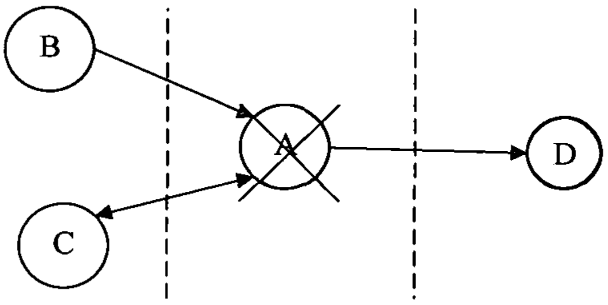


Figure 3-6: Blocking SCS and RCS

- **Repair Phase**

Two types of cells in the inflammatory phase of wound-healing process are responsible for repairing the injury site by producing some substances. These substances direct the next phase of the wound-healing process. Moreover, at the beginning of this phase, the body tries to remove the debris at the injury site in order to start repairing it.

In this phase, we aim to isolate the faulty component (remove debris) in order to start repairing it. There are two ways to repair the faulty component; either to mutate the component or to replicate it. In some cases the system performs the two repair plans; mutate and replicate. This phase contains:

• Module No.5: Repair Analyzer

The system needs to determine what action should be taken (for example to, mutate, replicate or mutate and replicate). The Repair Analyzer determines to replicate, mutate or mutate-replicate the component after receiving a message from the Fault Analyzer. The Repair Analyzer sends a message: (1) to the Replication Executor if the action is replicate, (2) to the Mutation Plan Generator if the action is mutate, (3) or to the Mutation Plan Generator and Replication Executor if the action is mutate-replicate.

• Module No.6: Mutation Plan Generator

This module generates the mutation plan against the faulty components. The generated mutation plan is based on the current state of the faulty component. The Mutation Plan Generator sends the mutation plan to the Mutation Plan Executor.

• Module No.7: Mutation Plan Executor

The task of this module is to execute the mutation plan which has been generated by the Mutation Plan Generator.

- **Repair Validation Phase**

In proliferation phase of wound-healing process, the cells start rebuilding the injury site by performing the mitotic activity. Here, we intend to make sure the chosen repair plan has been executed in a proper way.

- Module No.8: Mutation Plan Tester

This module tests whether the component after the mutation works. If the component after the mutation works properly, the Mutation Plan Tester sends a message to the Runtime Manager; otherwise it sends a message to the Mutation Plan Generator to choose other configuration plan.

- Module No.9: Replication Executor:

The Replication Executor module replicates the component after receiving a message from the Repair Analyzer.

- **Integration Phase**

In wound-healing process, the last phase tasks are remodeling the tissues and strengthen the scar at the injury site. In the same way, Integration phase task is to return the isolated component (healed component) to the system in a way that will not harm the running system.

- Module No.10: Runtime Manager

returns the healed component to back to the system by receiving two messages, one from the Mutation Plan Tester (indicates that the test result is positive) and the other from the Replication Executor (indicates that the replication completed.) it sends a notification message to the Fault Expansion Resistor to unblock the two sets of components in order to interact with the healed component.

3.4.2 The Formal Description

As software systems increase in size and complexity, using formal methods to specify and verify software systems are becoming more important. Modeling runs through many stages as specifications, model checking and analysis of properties, code generation, and execution of the code. Previous section introduced the specifications of self-healing software architecture. This section checks and analyzes the architecture by presenting a formal description of the architecture using set-theoretic. At the end of the description, we prove the associative and closure properties for our architecture. Table 3-2 describes the notations used in this chapter.

Table 3-2: Table of notations

Notation	Description
ρ	Functional Layer
μ	Healing Layer
A	Fault Detector
Σ	Fault Analyzer
ς$_i$	System's Component
κ(ς$_i$)	Constraints of component ς$_i$
Б	Fault Type
D	Fault Expansion Detector
R	Fault Expansion Resistor
SCS	Sending Components Set
RCS	Receiving Components Set
Z	Repair Analyzer
Y	Repair Plan
G	Mutation Plan Generator
Ω	Mutation Plan
M	Mutation Plan Executor
X	Replicate Executer
I	Run-Time Manager

3.4.2.1 Definition 1

A self-healing system S can be represented by a 2-tuple element $<\rho, \mu>$:

- ρ : is the functional layer of the system S that consists of a set of components and a logical framework: C is a set of components,
  $C = \{\varsigma_1, \varsigma_2 \dots \varsigma_n\}$.

  Each ς_i is a 4-tuple element: $<Fun_i, Interface_i, Info_i, Perf_i>$, where:

  n: is the number of the components in system S,
  $1 \leq i \leq n$
  Fun_i: is the function of component ς_i,
  $Interface_i$: is the interface of component ς_i,
  $Info_i$: is the information of component ς_i,
  $Perf_i$: is the performance of component ς_i.

- μ : is the healing layer of the system S represented by a 5-tuple element: <Monitor, Fault Control, Repair, Repair Validation, Integration>. Each of these elements is a finite set of modules. The numbers of modules in the finite set equals to the numbers of components in the healing layer. There is one module for each component.

  - Monitor: can be represented by 2-tuple element $<A, \Sigma>$:

    - A: is a finite set of sensors modules (or Fault Detectors).
      $$\forall \varsigma_i \in C \rightarrow \exists \alpha_i \in A.$$

      These modules analyze the state of the components at the run time (the time that the component receives input, executes process or sends output) using a set of constraints κ for each component at a particular time t.

$$\forall \varsigma_i \in C \rightarrow \exists \kappa(\varsigma_i)_t \subseteq \kappa(\varsigma_i).$$

The component ς_i is considered to be in its normal condition if α_i updates the component state as follows:

$$\forall \varsigma_i \in C, \exists\ t_j, t_{j+1}, \rightarrow \kappa(\varsigma_i)_{tj+1} \subseteq \kappa(\varsigma_i)_{tj}, \text{ where } \kappa(\varsigma_i)_{tj} = \kappa(\varsigma_i).$$

The component ς_i is considered to be in its abnormal condition if α_i updates the component state as follows:

$$\forall \varsigma_i \in C, \exists\ t_j, t_{j+1}, \rightarrow \kappa(\varsigma_i)_{tj+1} \not\subseteq \kappa(\varsigma_i)_{tj}, \text{ where } \kappa(\varsigma_i)_{tj} = \kappa(\varsigma_i).$$

If the component is in its abnormal condition, α_i moves the component to the next state with this input:

$$<\kappa(\varsigma_i)_{tj+1}, \kappa(\varsigma_i)_{tj}>.$$

The related components to the faulty component send/receive messages to/from the faulty component need to be blocked from sending and receiving messages. Therefore, α_i sends a blocking request to the Fault Expansion Detector.

<block>.

- Σ: is a finite set of modules (Fault Analyzers):

$$\forall \varsigma_i \in C \rightarrow \exists\ \sigma_i \in \Sigma.$$

The Fault Analyzer module σ_i analyzes the constraints of the faulty components c_i at a particular time t_j to find the type of fault $б_i$.

$$Б = \{б_1, б_2 \ldots б_m\},$$

$$\forall \varsigma_i \in C \rightarrow \exists\ \sigma_i \in \Sigma \xrightarrow{analyze} \kappa(\varsigma_i)_{tj+1} \not\subseteq \kappa(\varsigma_i)_{tj} \xrightarrow{find} б_i \in Б,$$

Then, the Fault Analyzer moves the faulty component to the next state with input:

$$<\kappa(\varsigma_i)_{tj+1}\ ,\ \kappa(\varsigma_i)_{tj},\ б_i>.$$

- Fault Control: can be represented by a 2-tuple element <D, R>:

    - D: a finite set of modules (Fault Expansion Detectors).

$$\forall \varsigma_i \in C \rightarrow \exists\ \eta_i \in D$$

The Fault Expansion Detector σ_i of a faulty component ς_i, creates two sets of components SCS, RCS:

SCS: is a set of components that send outputs to the faulty component ς_i.

$$SCS = \{\zeta_1, \zeta_2 \ldots \zeta_n\}, \text{ where } \zeta_1, \zeta_2 \ldots \zeta_n \in C.$$

RCS: is a set of components that receive inputs from the faulty component ς_i.

$$RCS = \{\xi_1, \xi_2 \ldots \xi_n\}, \text{ where } \xi_1, \xi_2 \ldots \xi_n \in C.$$

The Fault Expansion Detector moves the faulty component to the next state with input:

<SCS, RCS>

- R: a finite set of modules (Fault Expansion Resistors):

$$\forall \varsigma_i \in C \rightarrow \exists\ \varepsilon_i \in R.$$

The Fault Expansion Resistor receives the two sets of components, SCS and RCS, and blocks the two sets from sending/receiving messages to/from the faulty component.

$$\forall \varsigma_i \in C \rightarrow \exists\ \varepsilon_i \xrightarrow{block} SCS \wedge \varepsilon_i \xrightarrow{block} RCS, \text{ where } \varepsilon_i \in R.$$

After receiving a notification message from the Runtime Manager indicates that the faulty component is healed, the Fault Expansion Resistor unblocks the two sets of component.

$$\forall \varsigma_i \in C \rightarrow \exists\ \varepsilon_i \xrightarrow{unblock} SCS \wedge \varepsilon_i \xrightarrow{unblock} RCS, \text{ where } \varepsilon_i \in R.$$

- Repair: can be represented by 3-tuple element $<\Theta, \Psi, \Phi>$:

  - Θ: a finite set of modules (Repair Analyzers):

$$\forall \varsigma_i \in C \rightarrow \exists\ \theta_i \in \Theta.$$

The Repair Analyzer θ_i of a faulty component ς_i receive an input from the Fault Analyzer contains:

$$<\kappa(\varsigma_i)_{tj+1}\ ,\ \kappa(\varsigma_i)_{tj}, 6_i>.$$

Then the Repair Analyzer analyzes the state of the component and the fault type 6_i to find the suitable Repair Plan γ_j.

$Y = \{\gamma_1, \gamma_2, \gamma_3\}$, where

γ_1: Mutate,

γ_2: Replicate,

γ_3: Mutate-Replicate.

$$\forall\ \kappa(\varsigma_i)_{tj+1} \not\subset \kappa(\varsigma_i)_{tj} \rightarrow \exists\ 6_i \in Б\text{ , then,}$$

$$\forall\ 6_i \in Б \rightarrow \exists\ \gamma_n \in Y, : 1 \leq n \leq 3$$

After determining the suitable plan, the Repair Analyzer sends a notification message to the next state containing the appropriate plan:

$<\gamma_n>$.

- Ψ: a finite set of modules (Mutation Plan Generators):

$$\forall \varsigma_i \in C \rightarrow \exists \psi_i \in \Psi.$$

The Mutation Plan Generator ψ_i of a faulty component ς_i uses the current constraints of the component $\kappa(\varsigma_i)_{tj+1}$ and the type of the fault $б_i$ to find the suitable Mutation Plan ω_i.

$$\forall \ \kappa(\varsigma_i)_{tj+1} \not\subset \kappa(\varsigma_i)_{tj} \wedge \exists \ б_i \in Б \rightarrow \omega_i \in \Omega.$$

Then the Mutation Plan Generator ψ_i moves the faulty component to the next state with input:

$$<\omega_i>.$$

- Φ: a finite set of modules (Mutation Plan Executors):

$$\forall \varsigma_i \in C \rightarrow \exists \varphi_i \in \Phi.$$

The Mutation Plan Executor φ_i of a faulty component ς_i executes the Mutation Plan ω_i by performing the operation in definition 2, then notifies the Mutation Plan Tester to test the healed component:

<test>

- Repair Validation: can be represented by 2-tuple element <T, X>:
  - T: a finite set of modules (Mutation Plan Testers):

$$\forall \varsigma_i \in C \rightarrow \exists \tau_i \in T.$$

The Mutation Plan Tester τ_i tests whether the faulty component is healed. The Mutation Plan Tester τ_i achieves the test by sending a test data to the component. After receiving the output data, the τ_i checks the constraints of the component, if the constraints of the component after the test equals to the constraints of the component, then, the faulty component is healed. As a consequence, τ_i sends a notification message to the Run-Time Manager:

<succeed>,

Otherwise, τ_i sends a notification message back to the Mutation Plan Generator ψ_i to determine another mutation plan.

<failed>.

- X: a finite set of modules (Replicate Executors):

$$\forall \varsigma_i \in C \rightarrow \exists \chi_i \in X.$$

The Replicate Executor replicates the component by performing definition 3, and then sends a notification message to the Runtime Manager.

<replicated>

- Integration: can be represented by 1-tuple element <I>:
  - I: a finite set of modules (Runtime Managers).

$$\forall \varsigma_i \in C \rightarrow \exists \upsilon_i \in I.$$

The Runtime Manager υ_i returns the healed component ς_i to the running system. It sends an unblocking message to the Fault Expansion Resistor to unblock the two sets of component SCS, RCS.

3.4.2.2 Definition 2

An action in a self-healing system S with component ς_i is called mutate if the following conditions are satisfied:

1. $\forall \varsigma_i \in C, \exists \gamma_1 \in Y$ CurInfo $(\varsigma_i) \supseteq$ Cond $(\varsigma_i) \Rightarrow \varsigma_i \xrightarrow{mutate} \varsigma_i'$, where
2. $Fun(\varsigma_i) \subseteq Fun(\varsigma_i')$,
3. $Interface(\varsigma_i) \subseteq Interface(\varsigma_i')$,
4. $Perf(\varsigma_i) < Perf(\varsigma_i')$.

3.4.2.3 Definition 3

An action in a self-healing system S with component ς_i is called replicate if the following conditions are satisfied:

1. $\forall \varsigma_i \in C, \exists \gamma_2 \in Y$ CurInfo $(\varsigma_i) \supseteq$ Cond $(\varsigma_i) \Rightarrow \varsigma_i \xrightarrow{replicate} \varsigma_i'$, where
2. $Fun(\varsigma_i) = Fun(\varsigma_i')$,
3. $Interface(\varsigma_i) = Interface(\varsigma_i')$,
4. $Perf(\varsigma_i) = Perf(\varsigma_i')$.

3.4.2.4 Definition 4

An action in a self-healing system S with component ς_i is called mutate-replicate action if the following conditions are satisfied:

$\forall \varsigma_i \in C, \exists \gamma_1 \in Y$ CurInfo $(\varsigma_i) \supseteq$ Cond $(\varsigma_i) \Rightarrow \varsigma_i \xrightarrow{mutate} \varsigma_i'$

1. $Fun(\varsigma_i) \subseteq Fun(\varsigma_i')$
2. $Interface(\varsigma_i) \subseteq Interface(\varsigma_i')$
3. $Perf(\varsigma_i) < Perf(\varsigma_i')$

$\forall \varsigma_i' \in C, \exists \gamma_2 \in Y$ CurInfo $(\varsigma_i') \supseteq$ Cond $(\varsigma_i') \Rightarrow \varsigma_i' \xrightarrow{replicate} \varsigma_i''$,

1. Fun(ς_i') = Fun(ς_i'') ,

2. Interface(ς_i) = Interface(ς_i''),

3. Perf(ς_i') = Perf(ς_i'').

That means:

$$\varsigma_i \xrightarrow{mutate} \varsigma_i' \xrightarrow{replicate} \varsigma_i''$$

The system performs the mutate operation in order to heal the faulty component. Then, the system replicates the healed component.

3.4.3 Commutativity Property

Commutativity property means, within an expression two or more of the same commutativity operators in a row, the order of operations does not matter as long as the sequence of the operands is not changed. Next, we prove whether the commutativity property is satisfied in definition 2, 3, 4.

- Commutativity Property for Definition 2:

  In definition 2, the mutate operation, which is

$$ç_i \xrightarrow{mutate} ç_i',$$

does not satisfy the associate property because of the time dimension. In other words, the mutate operation is not reversible because of the time dimension; therefore the associate property is not satisfied.

- Commutativity Property for Definition 3

  Likewise, time dimension makes the replicate operation in definition 3 non reversible.

$$ç_i \xrightarrow{replicate} ç_i'$$

  As a consequence, the replicate operation does not satisfy the associate property.

- Commutativity Property for Definition 4

  Definition 4 which is the mutate-replicate operation consists of two parts, mutate and replicate. Here, we intend to prove the associate property of the mutate-replicate operation.

If the sequence of the operation is mutate then replicate,

$$\varsigma_i \xrightarrow{mutate} \varsigma_i' \xrightarrow{replicate} \varsigma_i''$$

After the faulty component ς_i is mutated, the resulting component ς_i' will have complete basic properties (Fun, Interface, Perf). The outcome of the fist part is:

$\text{Fun}(\varsigma_i) \subseteq \text{Fun}(\varsigma_i')$

$\text{Interface}(\varsigma_i) \subseteq \text{Interface}(\varsigma_i')$

$\text{Perf}(\varsigma_i) < \text{Perf}(\varsigma_i')$

The second part of the operation is to replicate the mutated component ς_i'. The outcome of the second part is:

$\text{Fun}(\varsigma_i') = \text{Fun}(\varsigma_i'')$,

$\text{Interface}(\varsigma_i) = \text{Interface}(\varsigma_i'')$,

$\text{Perf}(\varsigma_i') = \text{Perf}(\varsigma_i'')$.

At the end of the mutate-replicate operation, the resulting component will have the complete specifications of the system's component.

In the same way, if the sequence of the two part of the operation has changed, the resulting component will have the complete specifications of the system's component.

$$\varsigma_i \xrightarrow{replicate} \varsigma_i' \xrightarrow{mutate} \varsigma_i''$$

As a result, definition 4 satisfies the commutativity property.

3.4.4 Closure Property

The closure property means, the operation on members of the set produces a member of the set. Next, we prove whether the closure property is satisfied in definitions 2, 3, and 4.

- Closure Property for Definition 2

In definition 2, the mutate operation, which is

$$\varsigma_i \xrightarrow{mutate} \varsigma_i',$$

The result of this operation is:

$\mathrm{Fun}(\varsigma_i) \subseteq \mathrm{Fun}(\varsigma_i')$

$\mathrm{Interface}(\varsigma_i) \subseteq \mathrm{Interface}(\varsigma_i')$

$\mathrm{Perf}(\varsigma_i) < \mathrm{Perf}(\varsigma_i')$

After the faulty component ς_i is mutated, the resulting component ς_i' will have the complete specifications (Fun, Interface, Perf) of the system's component. Therefore, the mutate operation satisfies the closure property.

- Closure Property for Definition 3

The replicate operation in definition 3 is:

$$\varsigma_i \xrightarrow{replicate} \varsigma_i'$$

The result of this operation is:

$\mathrm{Fun}(\varsigma_i) = \mathrm{Fun}(\varsigma_i')$,

$\mathrm{Interface}(\varsigma_i) = \mathrm{Interface}(\varsigma_i')$,

$\mathrm{Perf}(\varsigma_i) = \mathrm{Perf}(\varsigma_i')$.

The replicate operation makes a copy of the system's component at a certain time. As a consequence, the resulting component will have the same specifications (Fun, Interface, Perf) of the system's component.

From definition 3, we can deduce that the replicate operation satisfies the closure property.

- Closure Property for Definition 4

To prove the closure property of definition 4, we need to perform the two part of this definition. The two part of mutate-replicate operation, definition 4, are:

1. Mutate:

$$\varsigma_i \xrightarrow{mutate} \varsigma_i',$$

The result of this part is:

$\text{Fun}(\varsigma_i) \subseteq \text{Fun}(\varsigma_i')$

$\text{Interface}(\varsigma_i) \subseteq \text{Interface}(\varsigma_i')$

$\text{Perf}(\varsigma_i) < \text{Perf}(\varsigma_i')$.

2. Replicate:

$$\varsigma_i' \xrightarrow{replicate} \varsigma_i''$$

The result of this part is:

$\text{Fun}(\varsigma_i') = \text{Fun}(\varsigma_i'')$,

$\text{Interface}(\varsigma_i) = \text{Interface}(\varsigma_i'')$,

$\text{Perf}(\varsigma_i') = \text{Perf}(\varsigma_i'')$.

At the end of the mutate-replicate operation, the resulting component will have the complete specifications of the system's component. We can deduce that the mutate-replicate operation satisfies the closure property.

Table 3-3 summarizes the commutativity and closure properties for the repair plans.

Table 3-3: Commutativity and Closure Properties for the Repair Plans

Property / **Operation**	**Commutativity**	**Closure**
mutate	X	✓
replicate	X	✓
mutate-replicate	✓	✓

3.5 Graphnet Model

Beside the set-theory definition, finite state machine (FSM) is usually encountered as graphical objects. In this section, we present a finite state machine called *graphnet*. Graphnet is presented to prove the functionality of our architecture. The graphnet model starts by describing the states of the system's component and the transitions between them.

- Definition 5

Let a deterministic graphnet is 5-tuple $\Psi = \{\Phi, \varsigma_i, I, \lambda, A,\}$ where

Φ: a finite set of component's state.

ς_i : start state of the component, where $\varsigma_i \in \Phi$.

I: a set of input alphabet.

λ: state transition function.

A: a set of accepting state where $A \in \Phi$, and $A = \{\varsigma_i\}$.

The accepting state of the component is pictured as double-circle and the non-accepting states of the component are pictured as single-circle. To prove the functionality of our architecture, the component must start from the accepting state, move through the other states, return to the starting and accepting state.

- **Functional layer**

In the functional layer, the component executes its operations without any disruptions. To execute these operations, the component moves through different states. Figure 3-7 illustrates the states of the component in the functional layer.

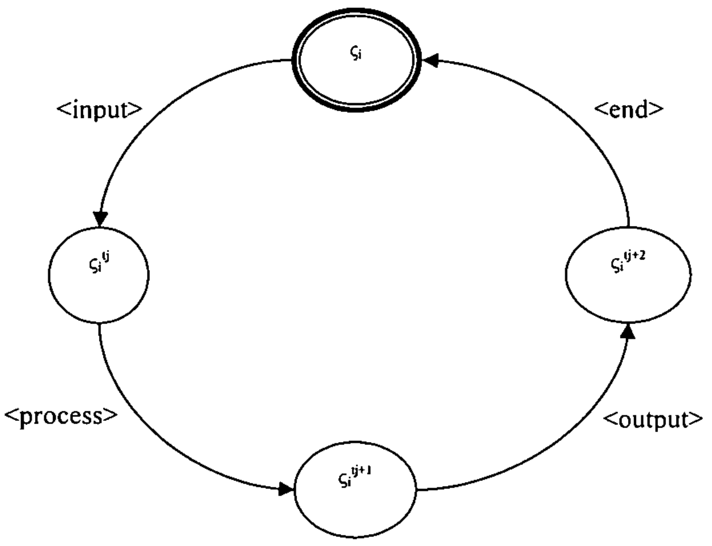


Figure 3-7: component's states in the functional layer

Path 1: ς_i - ς_i^{tj} - ς_i^{tj+1} - ς_i^{tj+2} - ς_i = input + process + output + end

In Figure 3.7 component ς_i receives an input at time t_j. Then, the component executes processes at time t_{j+1}, and sends an output at time t_{j+2}. At the end of the component execution, the component must return to the accepting state. This is the normal execution of the component in the functional layer. If any thing has gone wrong during the execution of the component, the component moves through the healing layer states.

- **Healing Layer:** in this layer, we aim to return the component to its accepting state. Next are the graphnets and descriptions of each phase in the healing layer. The component's accepting state is pictured as double-circle and the component's non-accepting states are pictured as single-circle.

- **Single-Fault:**

  The graphnet presents one component at a time. However, the architecture deals with single-fault. The healing layer creates one module for the faulty component in each state. This seems practical and easy to develop.

In Figure 3-8, path 1 is the normal execution of the system's component. Through this path, the component completes its operations without any interruptions. If there is any interruption during the execution of the component (receiving input, processing, or sending output), the component will move through path 2, 3, or 4. These paths will be completed through next phases.

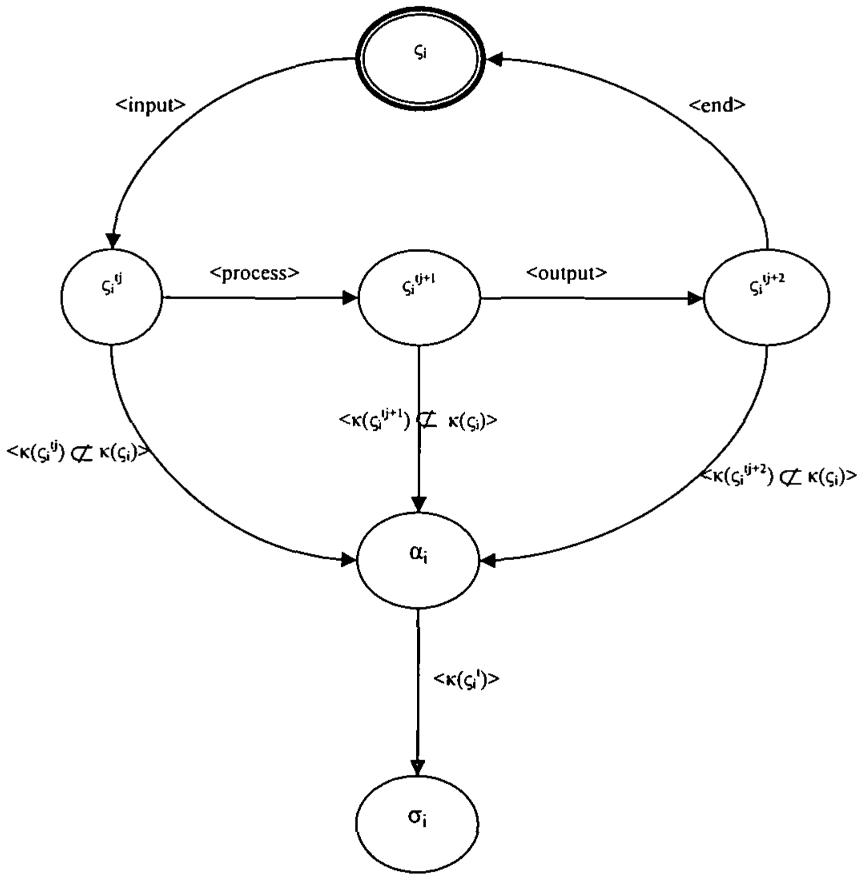
Each component in the systems has constraints which have been mentioned in the previous section. In the fault detection state σ_i, the Fault Detector Module checks the constraints of the component every time that the component moves from one state to another. If the constraints of the component at the specified time is not equal to the constraints of the component

$$<\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)>,$$
$$<\kappa(\varsigma_i^{tj+1}) \not\subset \kappa(\varsigma_i)>, \text{ or}$$
$$<\kappa(\varsigma_i^{tj+2}) \not\subset \kappa(\varsigma_i)>.$$

The Fault Detector Module moves the component state to fault analysis state. The transition function which leads to this state is:

$$\lambda(\kappa(\varsigma_i^t))$$

This is the constraints of the component at the time of the fault t. the Fault Analyzer Module receives the current state of the component (the constraints at time t) in order to analyze the state of the component and to determine the type of fault.



ς_i
<input>
<end>
ς_i^{tj}
<process>
ς_i^{tj+1}
<output>
ς_i^{tj+2}
$<\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)>$
$<\kappa(\varsigma_i^{tj+1}) \not\subset \kappa(\varsigma_i)>$
$<\kappa(\varsigma_i^{tj+2}) \not\subset \kappa(\varsigma_i)>$
α_i
$<\kappa(\varsigma_i^{t})>$
σ_i

Figure 3-8: Fault Detection graphnet

Path 1: ς_i - ς_i^{tj} - ς_i^{tj+1} - ς_i^{tj+2} - ς_i = input + process + output + end

Path 2: ς_i - ς_i^{tj} - α_i - σ_i ...= input + $<\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)>$ + $<\kappa(\varsigma_i^{t})>$ ...

Path 3: ς_i - ς_i^{tj} - ς_i^{tj+1} - α_i - σ_i ...= input + process + $<\kappa(\varsigma_i^{tj+1}) \not\subset \kappa(\varsigma_i)>$ + $<\kappa(\varsigma_i^{t})>$ ...

Path 4: ς_i - ς_i^{tj} - ς_i^{tj+1} - ς_i^{tj+2} - α_i - σ_i ... = input + process + output +$<\kappa(\varsigma_i^{tj+2}) \not\subset \kappa(\varsigma_i)>$ + $<\kappa(\varsigma_i^{t})>$ ...

In Figure 3-9, the component moves to the repair analysis state with the transition function

$$\lambda(\kappa(\varsigma_i^t), 6_i).$$

The input to this state is the current constraints of the component $\kappa(\varsigma_i^t)$ and the fault type 6_i which has been determined by the Fault Analyzer Module. In this state, the Repair Analyzer module determines the repair plan that should be taken according to the fault type. There are three plans that could be determined by the Repair Analyzer module; mutate, replicate, and mutate-replicate. The Repair Analyzer module sends the selected plan to the module that responsible for the specified plan.

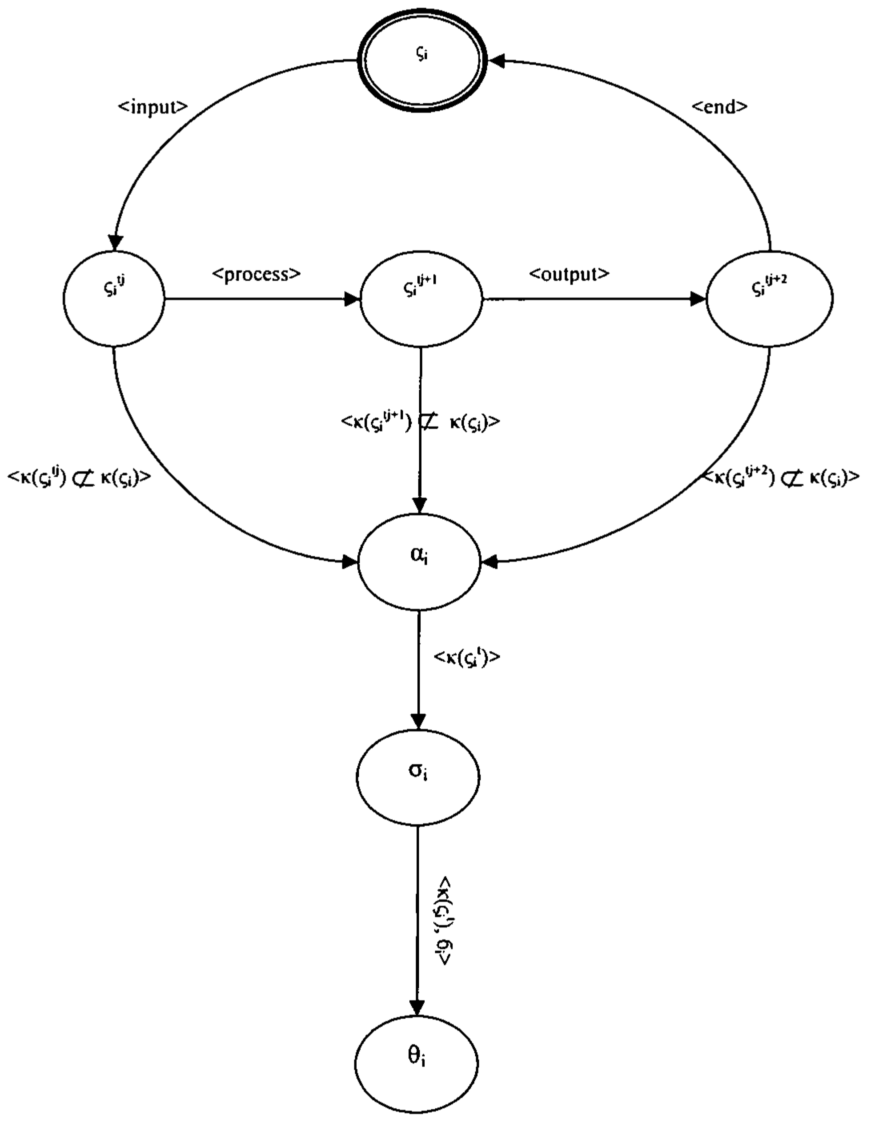

ς_i
<input>
<end>
ς_i^{tj}
<process>
ς_i^{tj+1}
<output>
ς_i^{tj+2}
$<\kappa(\varsigma_i^{tj+1}) \not\subset \kappa(\varsigma_i)>$
$<\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)>$
$<\kappa(\varsigma_i^{tj+2}) \not\subset \kappa(\varsigma_i)>$
α_i
$<\kappa(\varsigma_i^{t})>$
σ_i
$<\kappa(\varsigma_i^{t}), 6_i>$
θ_i

Figure 3-9: Repair Graphnet

Path 1: $\varsigma_i - \varsigma_i^{tj} - \varsigma_i^{tj+1} - \varsigma_i^{tj+2} - \varsigma_i$ = input + process + output + end

Path 2: $\varsigma_i - \varsigma_i^{tj} - \alpha_i - \sigma_i - \theta_i$ …= input + $<\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)>$ + $<\kappa(\varsigma_i^{t})>$ + $<\kappa(\varsigma_i^{t}), 6_i>$ …

Path 3: $\varsigma_i - \varsigma_i^{tj} - \varsigma_i^{tj+1} - \alpha_i - \sigma_i - \theta_i$ …= input + process + $<\kappa(\varsigma_i^{tj+1}) \not\subset \kappa(\varsigma_i)>$ + $<\kappa(\varsigma_i^{t})>$ + $<\kappa(\varsigma_i^{t}), 6_i>$ …

Path 4: $\varsigma_i - \varsigma_i^{tj} - \varsigma_i^{tj+1} - \varsigma_i^{tj+2} - \alpha_i - \sigma_i - \theta_i$ … = input + process + output + $<\kappa(\varsigma_i^{tj+2}) \not\subset \kappa(\varsigma_i)>$ + $<\kappa(\varsigma_i^{t})>$ + $<\kappa(\varsigma_i^{t}), 6_i>$ …

We described the states of the component until the repair plan state. In Figure 3-10 we introduce the mutate plan graphnet. We start the description of this figure from the repair plan state.

After the repair plan has determined by the Repair Analyzer module which is mutate, the component state moves to the mutate plan state by transition function

$$\lambda(<\gamma_j>).$$

In the mutate plan state ψ_i the Mutation Plan Generator module determine which mutate plan ω_i should be taken. By determining the mutate plan, the Mutation Plan Generator module moves the state of the component to the execute plan state by the transition function

$$\lambda(<\text{execute}, \omega_i>).$$

The Mutation Plan Executer module executes the selected mutate plan and moves the component to the testing state φ_i. The transition function is:

$$\lambda(<\text{test}>).$$

In the testing state, the Mutation Plan Tester module tests the mutated component to check whether the component mutated. The Mutation Plan Tester module moves the component to the integration state if the test succeeds,

$$\lambda(<\text{succeed}>).$$

Otherwise the Mutation Plan Tester module moves the component back to the mutate plan state.

$$\lambda(<\text{failed}>).$$

Once the component moves to the integration state, the Runtime Manager module moves the healed component back to the system (the accepting state).

$$\lambda(<\text{healed}>).$$

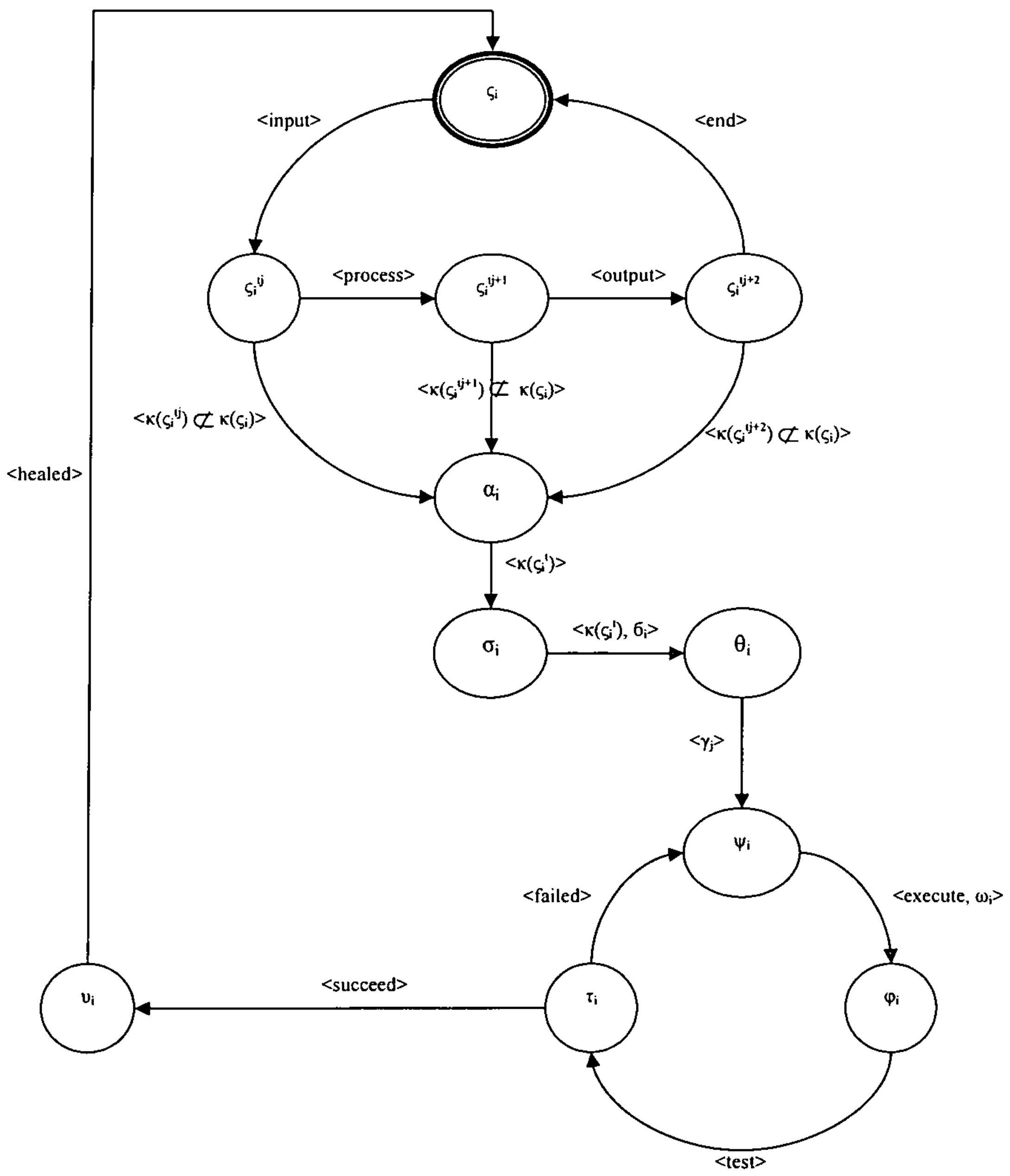


Figure 3-10: Mutate Plan Graphnet

Path 1: ς_i - ς_i^{tj} - ς_i^{tj+1} - ς_i^{tj+2} - ς_i = input + process + output + end

Path 2: ς_i - ς_i^{tj} - α_i - σ_i - θ_i - ψ_i - φ_i - τ_i - υ_i - ς_i = <input> + <$\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)$> + <$\kappa(\varsigma_i^{t})$> + <$\kappa(\varsigma_i^{t})$, 6_i> + <γ_j> + <test> + <ω_i> + <succeed> + <healed>

Path 3: ς_i - ς_i^{tj} - ς_i^{tj+1}- α_i - σ_i - θ_i - ψ_i - φ_i - τ_i - υ_i - ς_i = <input> + <process> + <$\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)$> + <$\kappa(\varsigma_i^{t})$> + <$\kappa(\varsigma_i^{t})$, 6_i> + <γ_j> + <test> + <ω_i> + <succeed> + <healed>

Path 4: ς_i - ς_i^{tj} - ς_i^{tj+1}- ς_i^{tj+2} - α_i - σ_i - θ_i - ψ_i - φ_i - τ_i - υ_i - ς_i = <input> + <process> + <output> + <$\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)$> + <$\kappa(\varsigma_i^{t})$> + <$\kappa(\varsigma_i^{t})$, 6_i> + <γ_j> + <test> + <ω_i> + <succeed> + <healed>

Figure 3-11 illustrates the replicate plan graphnet. The replicate plan is to make a copy the component. The Replicate Executor module makes a copy of the component and moves the component state to the integration state.

$$\lambda(\text{<replicated>}).$$

Once the component moves to the integration state, the Runtime Manager module moves the healed component back to the system (the accepting state).

$$\lambda(\text{<healed>}).$$

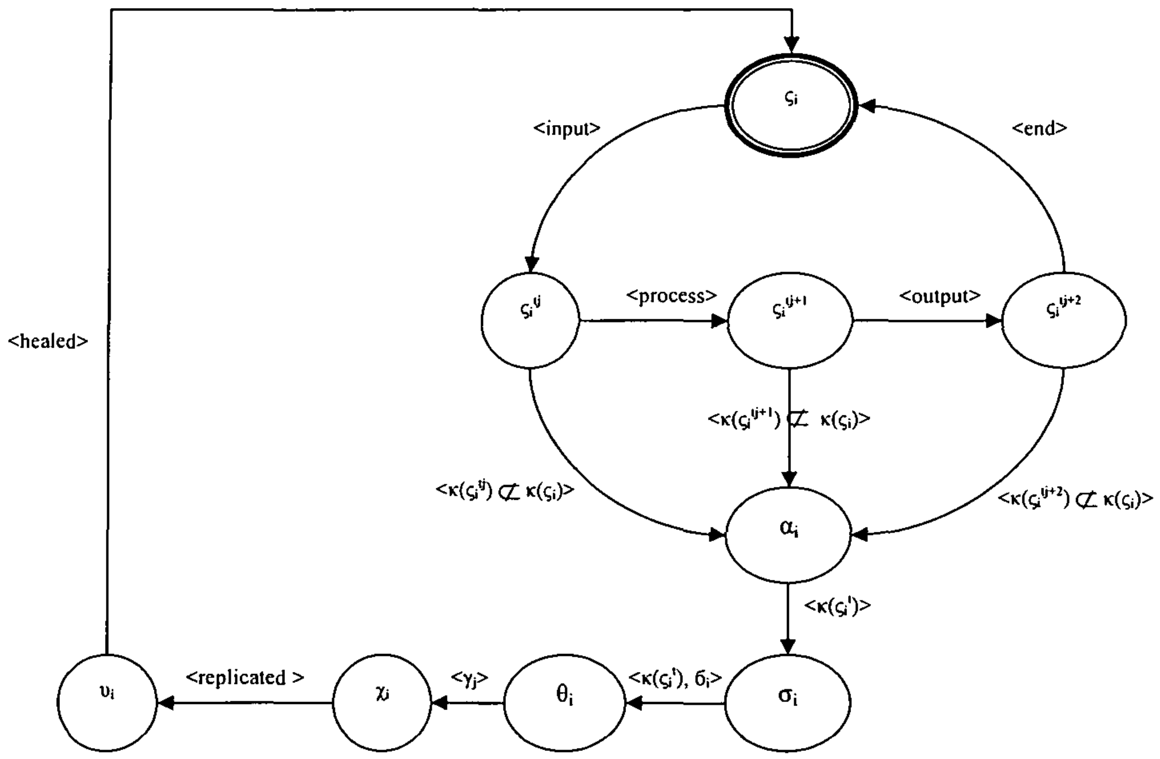

ς_i
<input>
<end>
ς_i^{tj}
<process>
ς_i^{tj+1}
<output>
ς_i^{tj+2}
<healed>
$<\kappa(\varsigma_i^{tj+1}) \not\subset \kappa(\varsigma_i)>$
$<\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)>$
$<\kappa(\varsigma_i^{tj+2}) \not\subset \kappa(\varsigma_i)>$
α_i
$<\kappa(\varsigma_i')>$
υ_i
<replicated >
χ_i
$<\gamma_j>$
θ_i
$<\kappa(\varsigma_i'), б_i>$
σ_i

Figure 3-11: Replicate Plan Graphnet

Path 1: $\varsigma_i - \varsigma_i^{tj} - \varsigma_i^{tj+1} - \varsigma_i^{tj+2} - \varsigma_i$ = input + process + output + end

Path 2: $\varsigma i - \varsigma_i^{tj} - \alpha_i - \sigma_i - \theta_i - \chi_i - \upsilon_i - \varsigma_i$ = input + $<\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)>$ + $<\kappa(\varsigma_i^{t})>$ + $<\kappa(\varsigma_i^{t}),$ $б_i>$ + $<\gamma j>$ + <replicated > + <healed>

Path 3: $\varsigma i - \varsigma_i^{tj} - \varsigma_i^{tj+1} - \alpha_i - \sigma_i - \theta_i - \chi_i - \upsilon_i - \varsigma_i$ = input + process + $<\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)>$ + $<\kappa(\varsigma_i^{t})>$ + $<\kappa(\varsigma_i^{t}), б_i>$ + $<\gamma j>$ + <replicated > + <healed>

Path 4: $\varsigma i - \varsigma_i^{tj} - \varsigma_i^{tj+1} - \varsigma_i^{tj+2} - \alpha_i - \sigma_i - \theta_i - \chi_i - \upsilon_i - \varsigma_i$ = input + process + output + $<\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)>$ + $<\kappa(\varsigma_i^{t})>$ + $<\kappa(\varsigma_i^{t}), б_i>$ + $<\gamma j>$ + <replicated > + <healed>

When the Repair Analyzer module determines to mutate-replicate the component, the component moves through the mutate plan states first, then through the replicate plan states. Figure 3.12 illustrates the mutate-replicate plan graphnet.

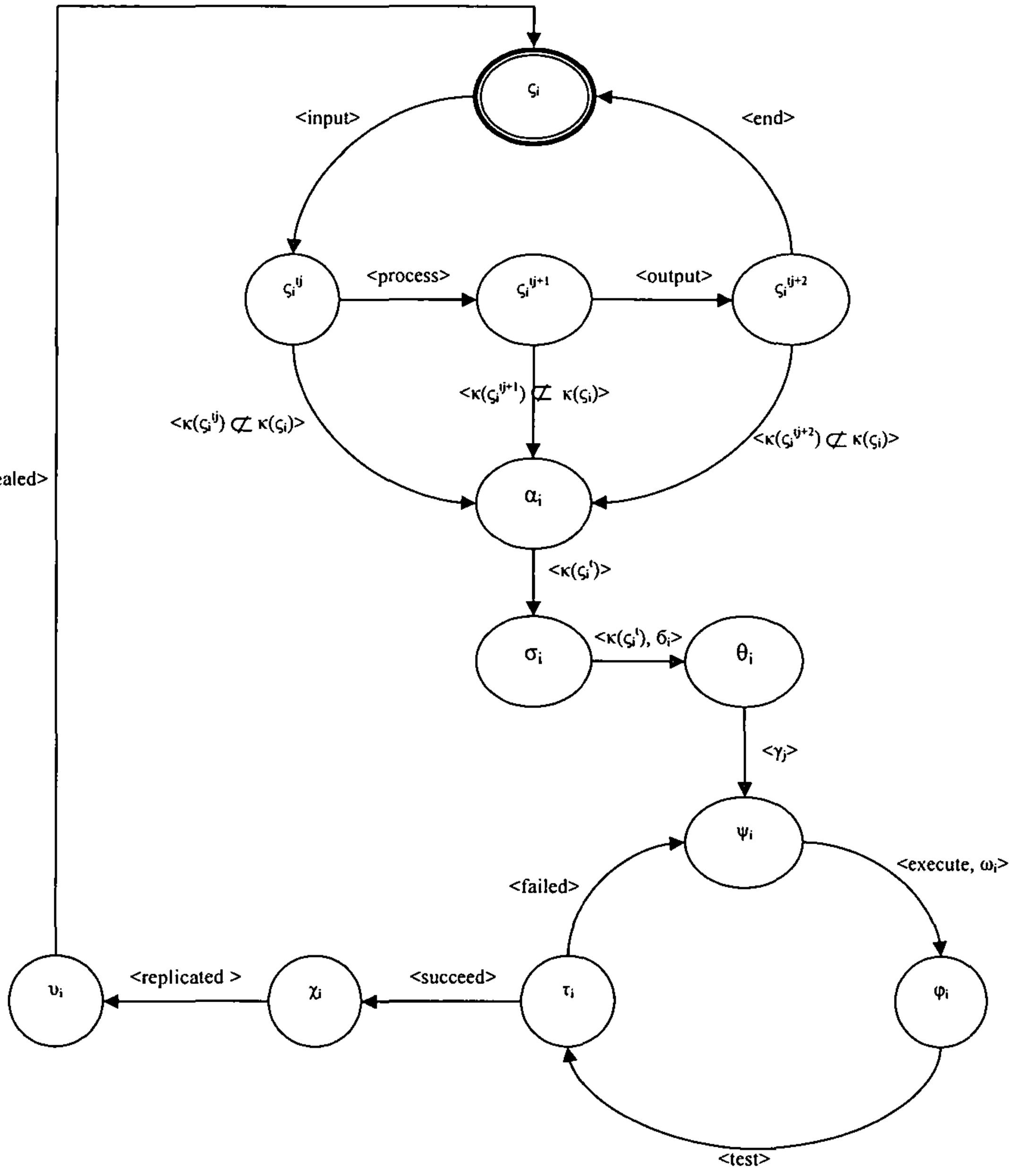


Figure 3-12: Mutate-Replicate Plan Graphnet

Path 1: ς_i - ς_i^{tj} - ς_i^{tj+1} - ς_i^{tj+2} - ς_i = input + process + output + end

Path 2: ς_i - ς_i^{tj} - α_i - σ_i - θ_i - ψ_i - φ_i - τ_i - χ_i - υ_i - ς_i = input + $<\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)>$ + $<\kappa(\varsigma_i^{t})>$ + $<\kappa(\varsigma_i^{t}), б_i>$ + $<\gamma_j>$ + <test> + $<\omega_i>$ + <succeed> + <replicated> +<healed>

Path 3: ς_i - ς_i^{tj} - ς_i^{tj+1}- α_i - σ_i - θ_i - ψ_i - φ_i - τ_i - χ_i - υ_i - ς_i = input + process + $<\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)>$ + $<\kappa(\varsigma_i^{t})>$ + $<\kappa(\varsigma_i^{t}), б_i>$ + $<\gamma_j>$ + <test> + $<\omega_i>$ + <succeed> + <replicated> + <healed>

Path 4: ς_i - ς_i^{tj} - ς_i^{tj+1}- ς_i^{tj+2} - α_i - σ_i - θ_i - ψ_i - φ_i - τ_i - χ_i - υ_i - ς_i = input + process + output + $<\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)>$ + $<\kappa(\varsigma_i^{t})>$ + $<\kappa(\varsigma_i^{t}), б_i>$ + $<\gamma_j>$ + <test> + $<\omega_i>$ + <succeed> + <replicated> + <healed>

- **Concurrent-Faults**

The architecture can handle single-fault as well as concurrent-faults. To handle concurrent-faults, the system creates one module for each component. See Figure 3-13, 3-14, and 3-15. In each state of the components, the system creates a set of modules in order to handle concurrent-faults. The number of modules in each state equals to the number of faulty components.

For example, in java programming, the set of modules can be done by using threads. Java virtual machine (JVM) allows the application to have multiple threads of execution running concurrently. By using advanced programming language we can develop a self-healing system which can handle single and concurrent faults.

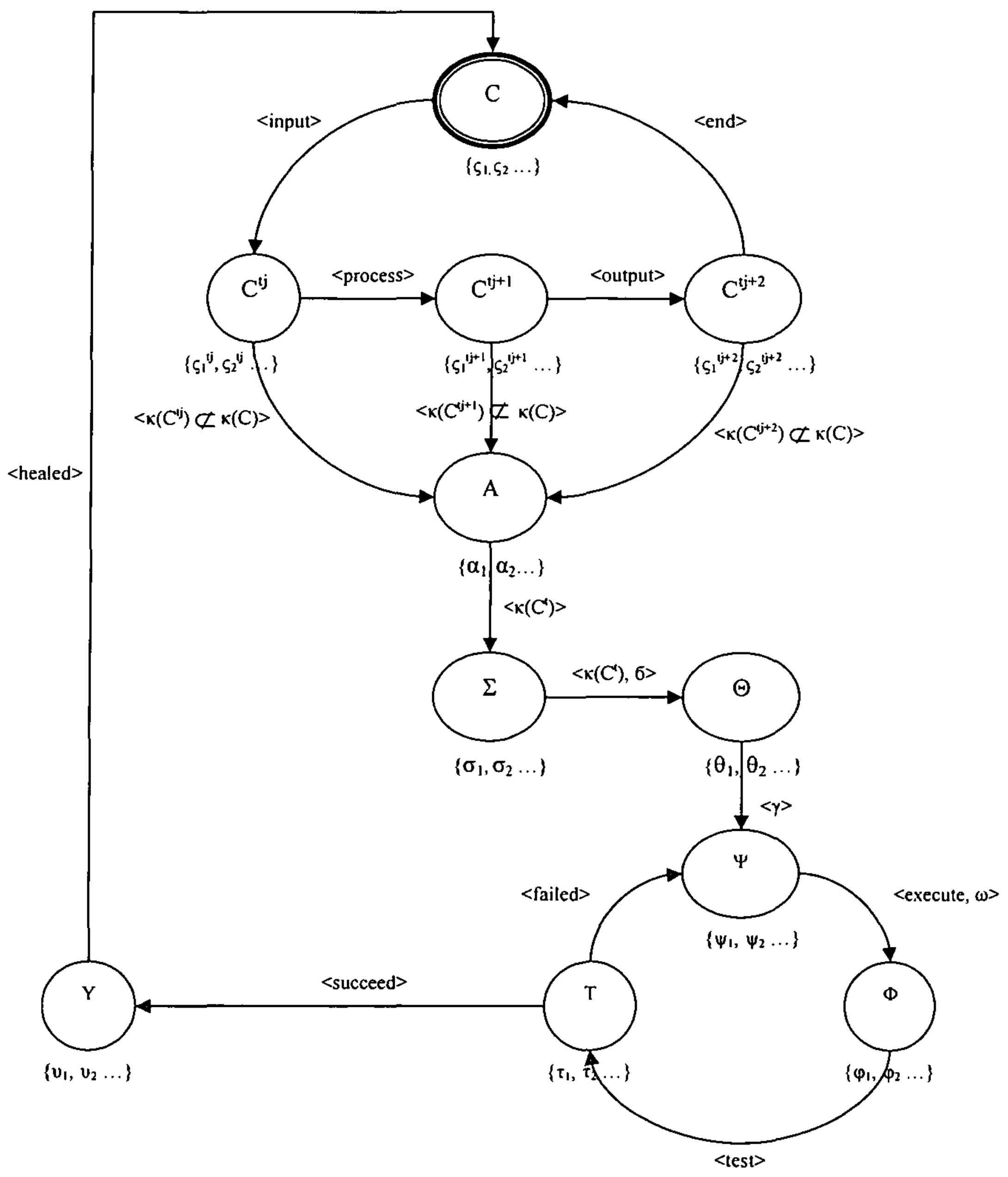


Figure 3-13: Mutate Plan Graphnet for Concurrent-Faults

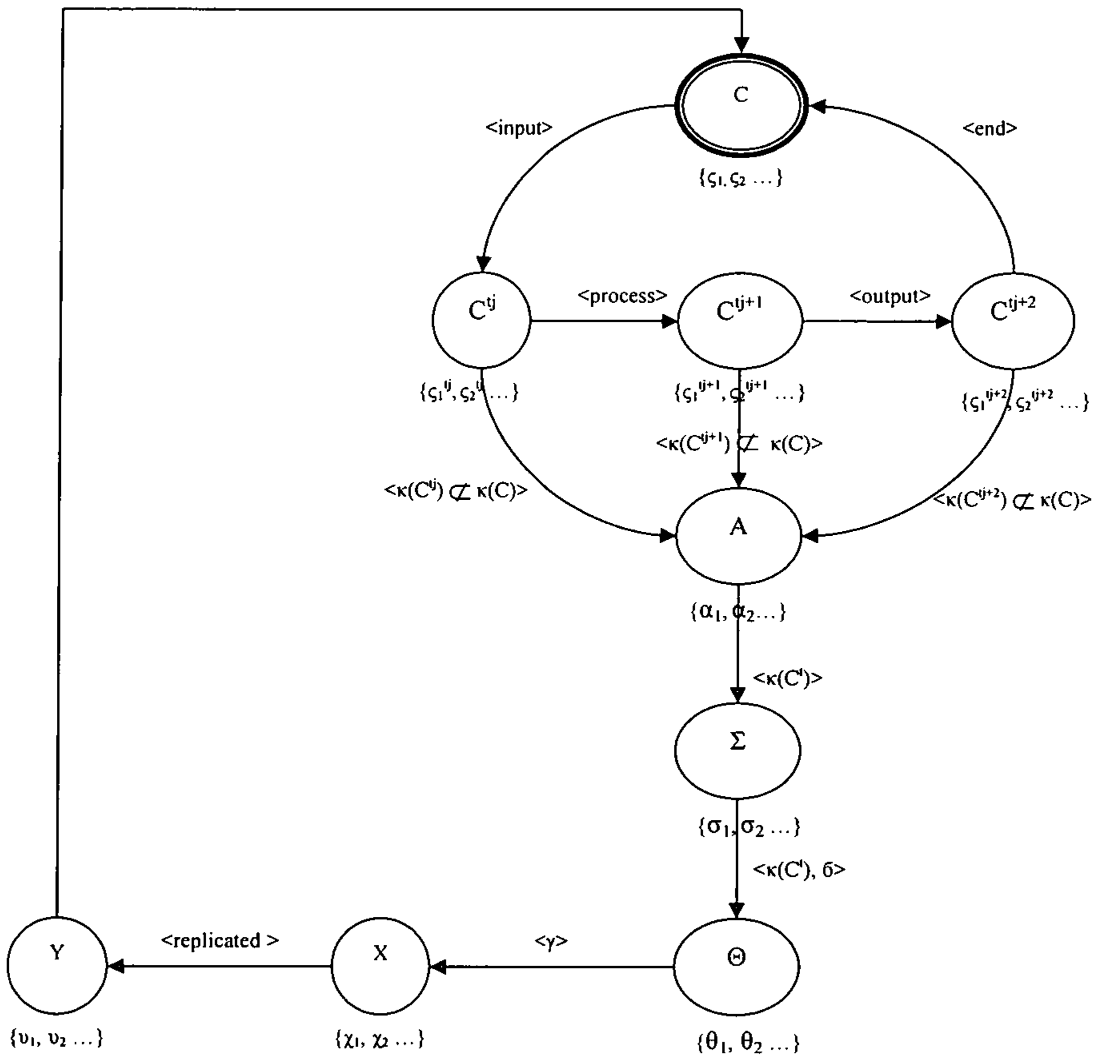


Figure 3-14: Replicate Plan Graphnet for Concurrent-Faults

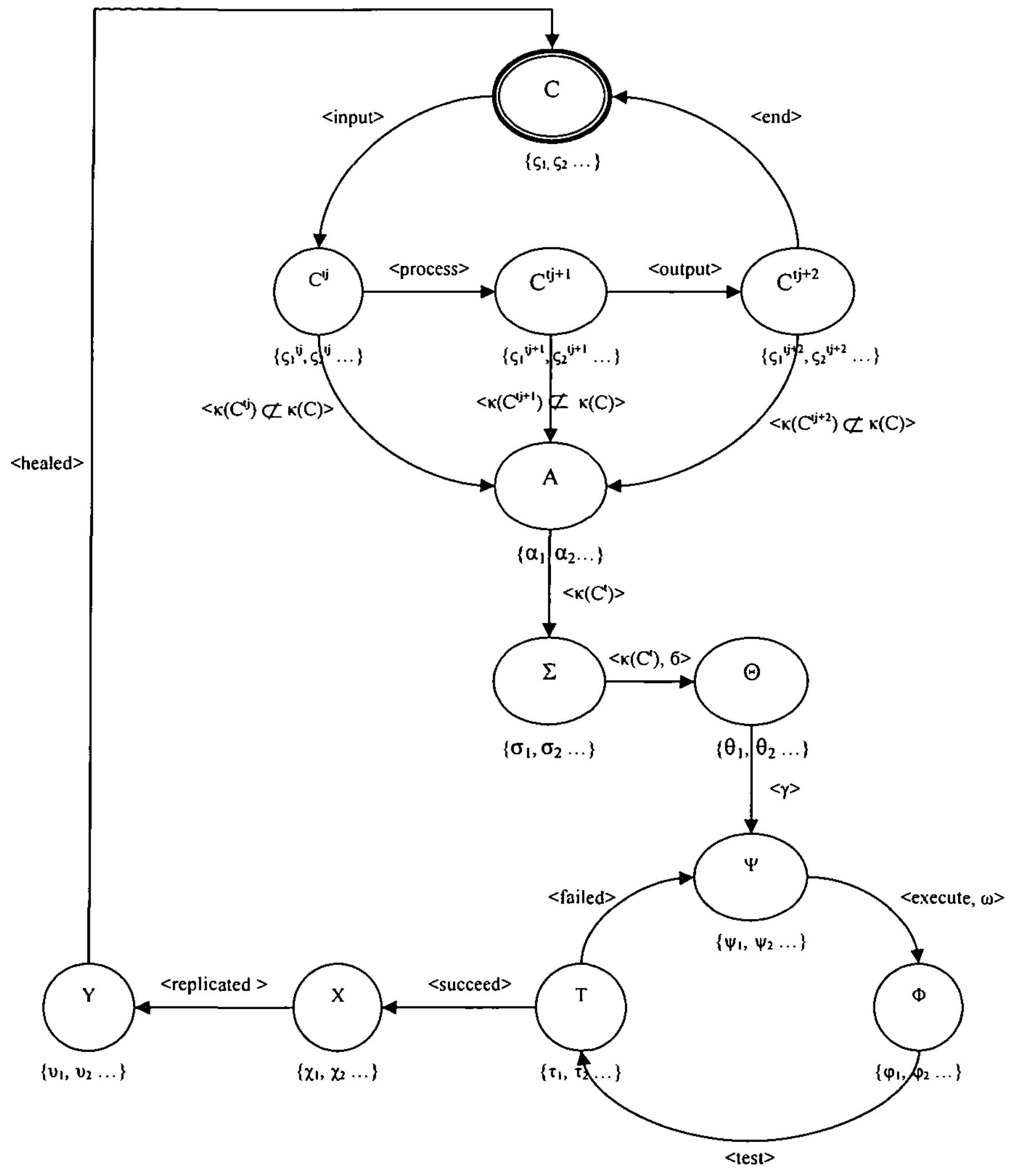


Figure 3-15: Mutate-Replicate Plan Graphnet for Concurrent-Faults

3.6 Conclusion

This chapter has introduced self-healing software system architecture based on the wound-healing process. Theoretical and formal descriptions of the architecture have been presented. To prove the functionality of the architecture, closure and commutativity properties have been discussed. The end of this chapter has presented finite state machine called graphnet which describes that states of the architecture during single and concurrent faults.

CHAPTER FOUR: SELF-HEALING SOFTWARE PROTOTYPE

4.1 Introduction

Self-healing applications should be able to recover from potential faults (Jeffery & David, 2003) and should continue to work smoothly without human intervention. In this chapter, we applied our architecture into two case studies. We simulate these case studies by developing two Java applications. The first application is a simple Java application that retrieves and sends data to a file. The second application is a simulation of Automated Teller Machine (ATM) system which provides the basic financial transactions. The chapter begins by a brief description of the Java language and presents the concepts of exceptions in Java. This chapter also presents the class diagram of the developed software. At the end of this chapter, the output of the developed applications is introduced.

4.2 Java Language

Java is a high-level programming language developed by Sun Microsystems. Similar to C++, Java is an object-oriented language. Moreover, Java is a general purpose programming language with a number of features that make the language able to simplify and eliminate common errors that continuously appear in other language.

Java supports object-oriented programming which is the way of programming that programmers define the data type of a data structure as well as the operations or functions that can be applied to the data structure. The data structure becomes an object which includes both data and operations. Furthermore, programmers can create the relationships between objects. The concepts of object-oriented programming in Java is applied by building blocks that contains data type and operations, this block is called *class*. The concepts of Java class provide these benefits:

- defining subclasses of data object that share same or some of the main class characteristics. These benefits simplify data analysis, reduce development time, and ensure more accurate coding.

- hiding class data and providing greater system security and avoids data corruption.

- the class can be used by the program that is initiated for as well as by other object-oriented programs.

- programmers can create any new data types that are not defined in Java language for any purposes.

Using these features, we implemented self-healing software based on our architecture using Java. Each module in the proposed self-healing architecture is represented as a class. Next section introduces the main classes, describes their tasks, and presents the relationships between them.

4.3 Architecture Prototype

In this section, we introduce the Unified Modeling Language (UML) Class Diagram for our prototype (Figure 4-1). The diagram contains classes that represent the functional and healing modules of the proposed self-healing software architecture.

The functional layer consists of components which are represented as classes. Each component provides particular services. The program user access this services via graphical user interface. Some components provide their services without user graphical interface. They provide their services to other components. The normal situation of the functional layer is:

- each component provides its services (and interface if any) without any disruption.

- All the components interact with each other without any disruption.

In the healing layer, each component associates with a set of modules. These modules are represented as classes.

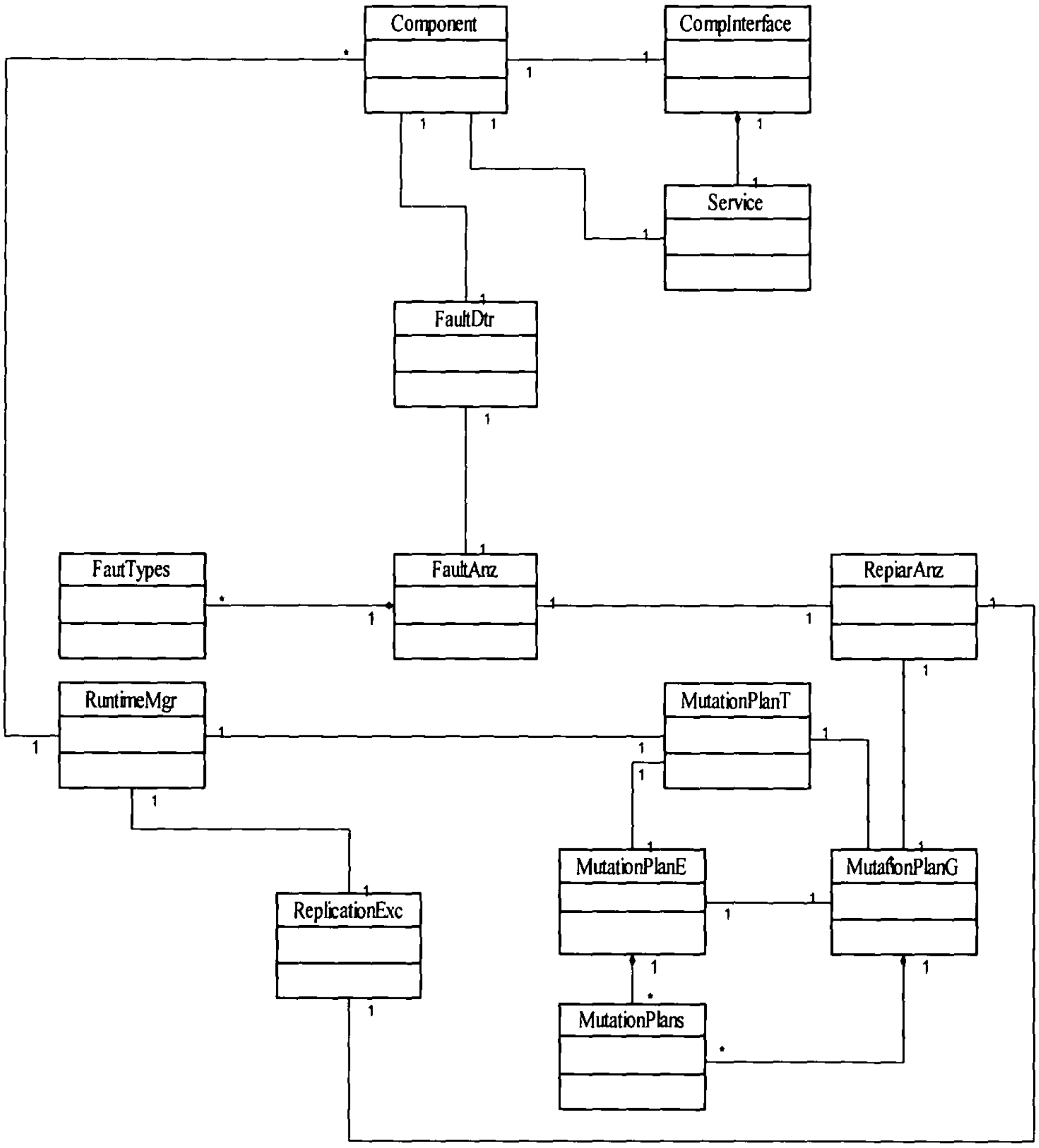


Figure 4-1: UML Class Diagram for self-healing Software System

Each class in the class diagram has a particular task. Table 4-1 illustrates the tasks that have been identified to each class. These tasks help the creation of the conceptual model which contains the relation between these classes.

Table 4-1: Self-Healing Software (Main Classes)

Class	Description
Component	Component of the system
Service	Services delivered by the system component
Interface	The component's interface
FaultDtr	Monitors the execution of the component
FaultAnz	Analyzes changes (faults) in the component
FaultTypes	Contains possible faults of the component
RepairAnz	Determines the repair plan that must be taken
MutationPlanG	Determines the mutation plan that must be applied
MutationPlanE	Executes the chosen mutation plan
MutationPlanT	Tests the component after mutation
MutationPlans	Contains possible mutation plans
ReplicateExc	Replicates the component
RuntimeMgr	Returns the component to the system

Figure 4-2 illustrates the sequence diagram. In this diagram we introduce the message passing between the system modules in order to complete the healing process.

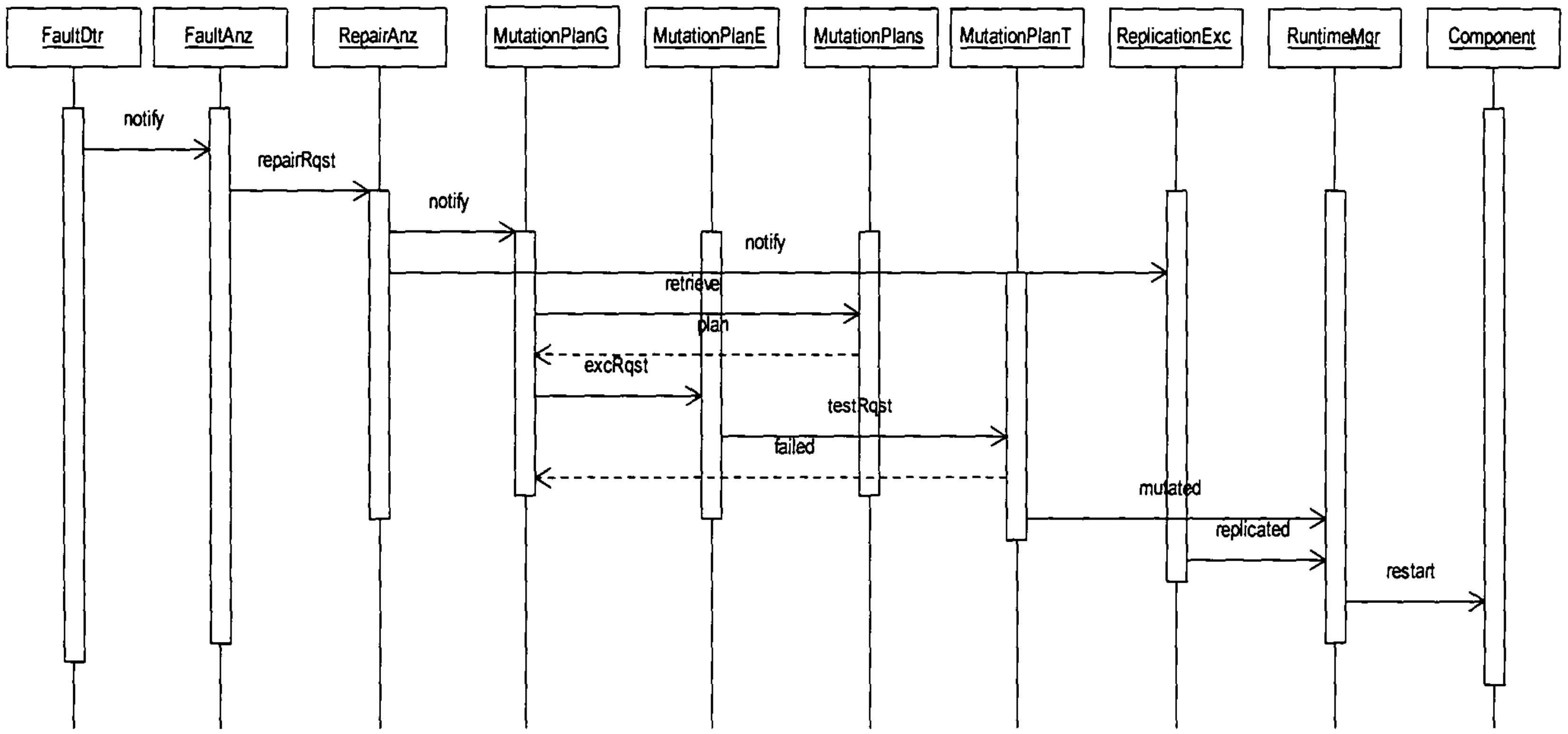


Figure 4-2: Sequence Diagram for Self-Healing Software System

4.4 Example of Faults in Java Objects

This section presents Java program that has the ability to returns to its normal execution after runtime error has occurred. We applied the proposed architecture into this application by creating one class for each module. Next is the description of the application. Java language provides a technique to detect common errors that appears during the runtime execution. This technique called Exception Handling.

4.4.1 Exceptions in Java

Exception is any abnormal, unexpected events or extraordinary conditions that may occur during runtime execution. Java Exceptions are basically Java objects. No Project can ever escape a Java error exception.

Java exception handling is used to handle error conditions in a program systematically by taking the necessary action. Exception handlers can be written to catch a specific exception such as Number Format exception or an entire group of exceptions by using generic exception handlers. Any exceptions not specifically handled within a Java program are caught by the Java run time environment.

We use this concept to detect faults that occur during the execution of Java objects. Java can detect the abnormal condition but the thing is, how to recover the system from this fault.

4.5 Application Task

The application provides GUI to the user who tries to access some files on the system. The system has number of components that communicate with each other in order deliver the system's services to the user (see Figure 4-3). For example, the user tries to access file called FileA. To deliver this service, the system provides:

- graphical user interface to access the service through it,
- components to deliver the services,
- and interaction between these components to complete the tasks.

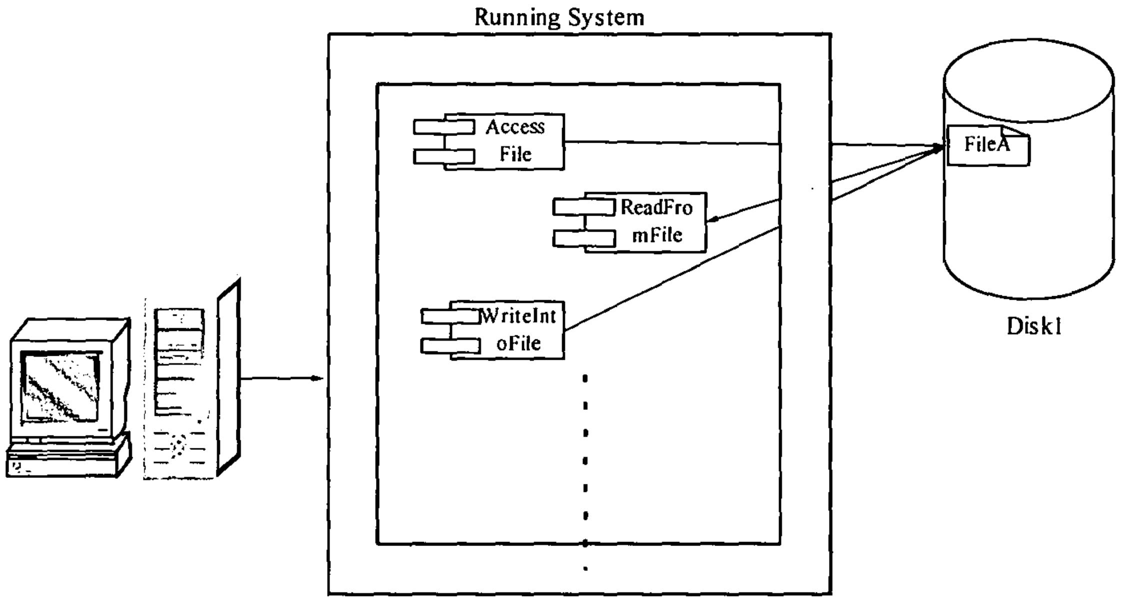
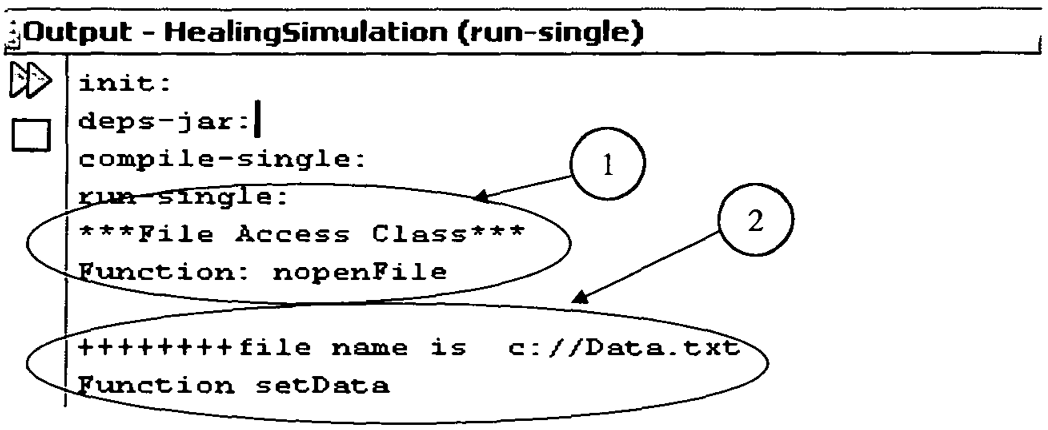


Figure 4-3: Accessing FileA from the Running System

In Figure 4-4, assume that, the user tries to save some data into Data.txt File. AccessFile object WriteIntoFile object will finish this task. AccessFile object accesses Data.txt and open it (output 1). Then, it sends the file pointer to the WriteIntoFile object. WriteIntoFile object store the data into Data.txt and closes the file (output 2).



```
init:
deps-jar:
compile-single:
run-single:
***File Access Class***
Function: nopenFile

++++++++file name is  c://Data.txt
Function setData
```

Figure 4-4: Save Data into Data.txt

Likewise, if the user tries to retrieve some data from Data.txt File, AccessFile object accesses Data.txt and open it. Then, it sends the file pointer to the ReadFromFile object. ReadFromFile object retrieves the data from Data.txt and closes the file.

If Data.txt file does not exist in the system, AccessFile object will through an exception. Therefore, the user will not be able to store or retrieve data from the system. In order to fix this problem, every time the user modifies Data.txt, the system makes a backup file from Data.txt. If Data.txt has been deleted or corrupted, the running system will not be able to access it. In this case, the service will be denied to the user.

To fix this problem, the healing part of the system tries to return the service to the user without any human intervention. The healing part suggests a plan to return the deleted or corrupted. The plan is to make a copy from the backup file in the same deleted or corrupted file location (disk).

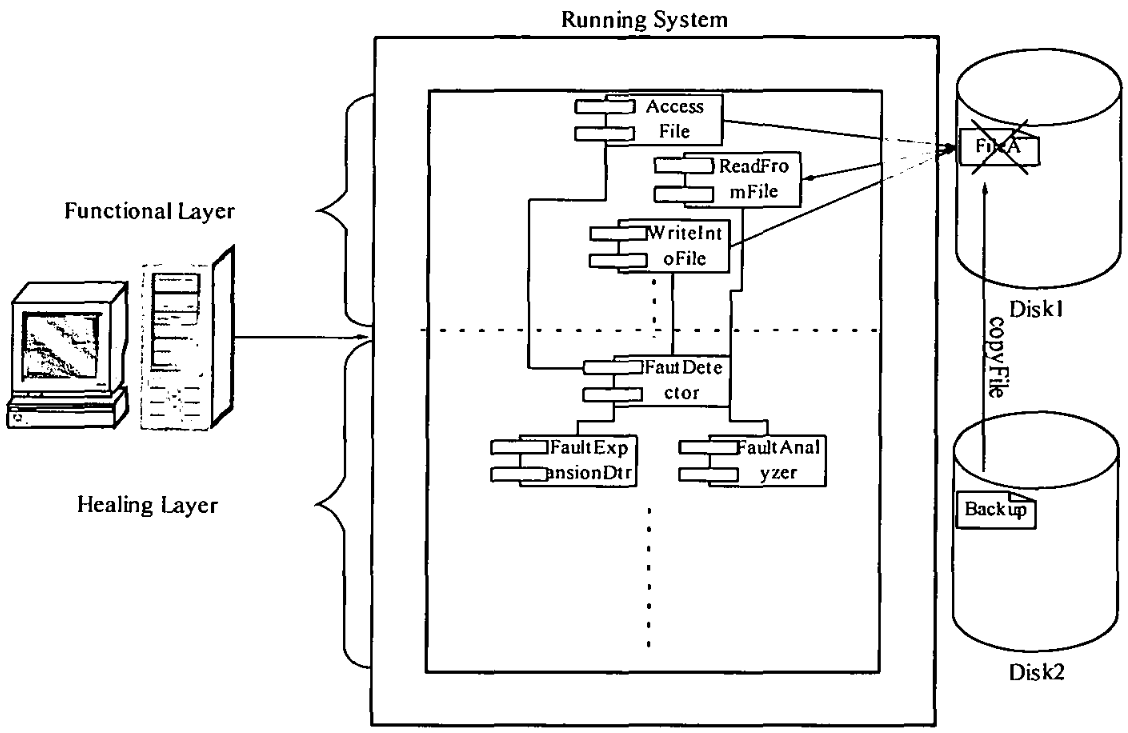


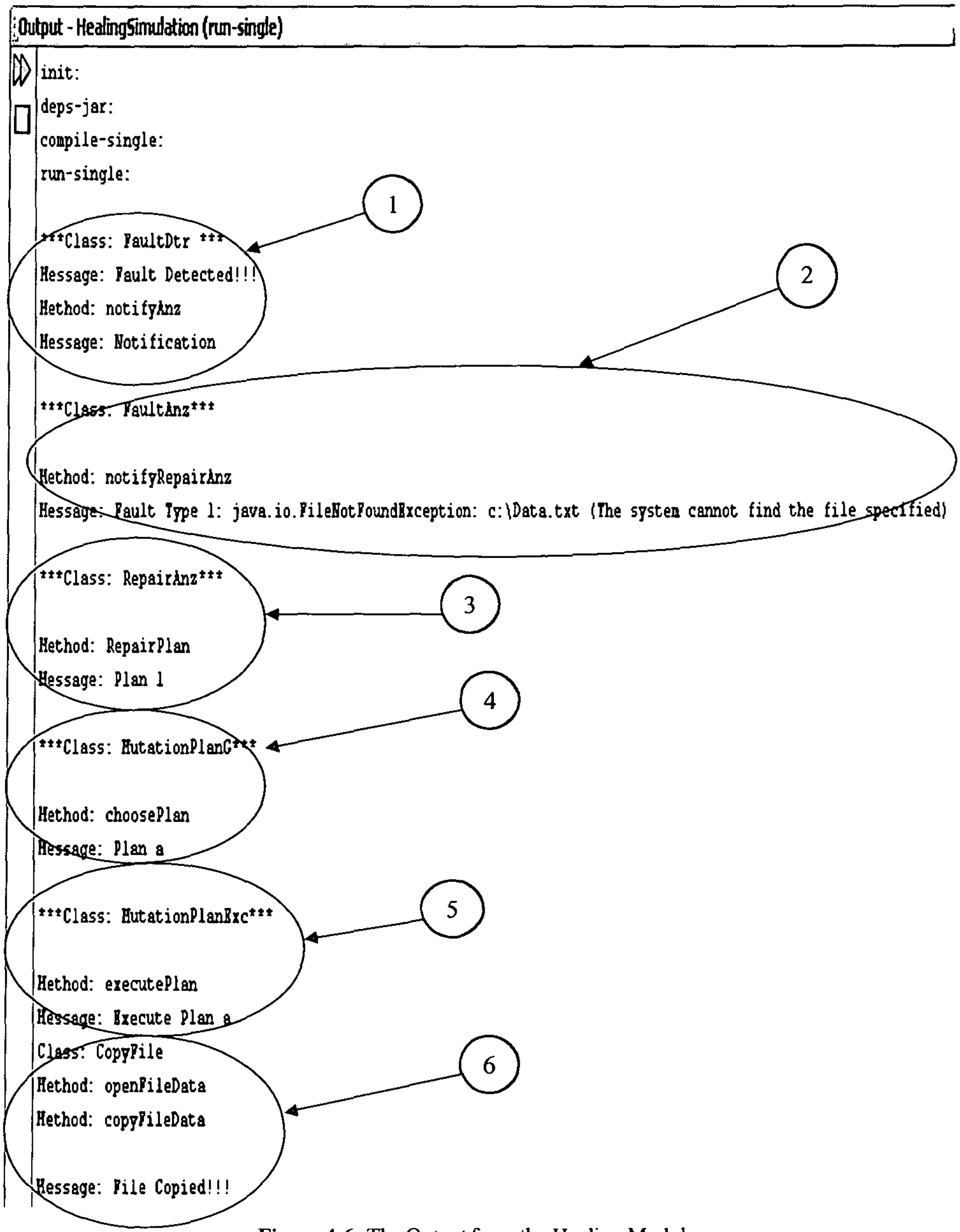
Figure 4-5: Copying Backup File

The healing process moves through the steps that exist in the proposed architecture. Each module in the healing process performs its task and presents an output showing its operations. As we mentioned earlier in this chapter, the healing modules are represented by classes. When a healing module is needed, an object from the healing module is created. The object is destroyed after completing its task.

Figure 4-6 shows the output of each module. Next is the table of the circles area in Figure 4-6 and the healing object (module) that responsible for it.

Table 4-2: Output Messages from Healing Modules

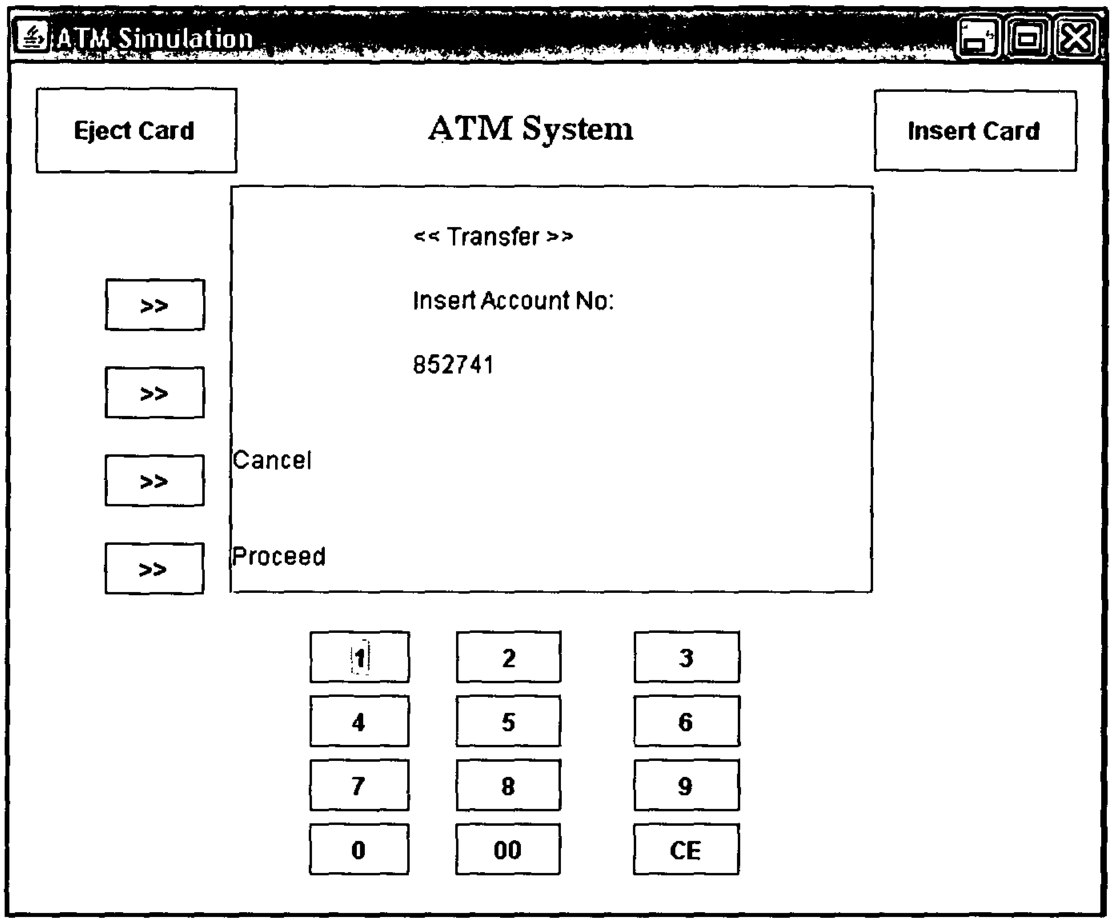
Area	Module
1	Fault Detector
2	Fault Analyzer
3	Repair Analyzer
4	Mutation Plan Generator
5	Mutation Plan Executer
6	Mutation Plan A Executer



```
init:
deps-jar:
compile-single:
run-single:

***Class: FaultDtr ***
Message: Fault Detected!!!
Method: notifyAnz
Message: Notification

***Class: FaultAnz***

Method: notifyRepairAnz
Message: Fault Type 1: java.io.FileNotFoundException: c:\Data.txt (The system cannot find the file specified)

***Class: RepairAnz***

Method: RepairPlan
Message: Plan 1

***Class: MutationPlanC***

Method: choosePlan
Message: Plan a

***Class: MutationPlanExc***

Method: executePlan
Message: Execute Plan a
Class: CopyFile
Method: openFileData
Method: copyFileData

Message: File Copied!!!
```

Figure 4-6: The Output from the Healing Modules

Moreover, the customer can transfer money from her/his account to any other account liked to the bank. In Figure 4-13, the system asks the customer to enter the account number that she/he wants to transfer the money to.



<< Transfer >>

Insert Account No:

852741

Cancel

Proceed

Figure 4-13: Transfer Transaction: Destination Account

4.6 Case Study: ATM System

Automated Teller Machine (ATM) is a telecommunications device that computerizes the financial transactions in a financial institution and allows the customer to access these transactions in a public space without the need for human clerk (Rog'erio, Jon, Modupe & Simon, 2007). This section introduces software system that simulates the Automated Teller Machine (ATM).

We developed a simple ATM application using Java. The ATM application provides the basic financial transactions. The customers can check their accounts, withdraw cash, and transfer money to other customers. The screen shots of the basic financial transactions of the ATM system are introduced in this chapter.

The ATM system services one customer at a time. The customer needs to insert a special plastic card into an ATM card reader. After inserting the plastic card into the ATM card reader, the customer needs to enter Personal Identification Number (PIN) using a keypad. The PIN will be transmitted to the bank central system. This number prevents unauthorized persons from performing transactions. If the PIN code is correct, the customer will be able to perform one or more financial transactions. The card will be inside the card reader until the customer indicates that they desires to exit from the system.

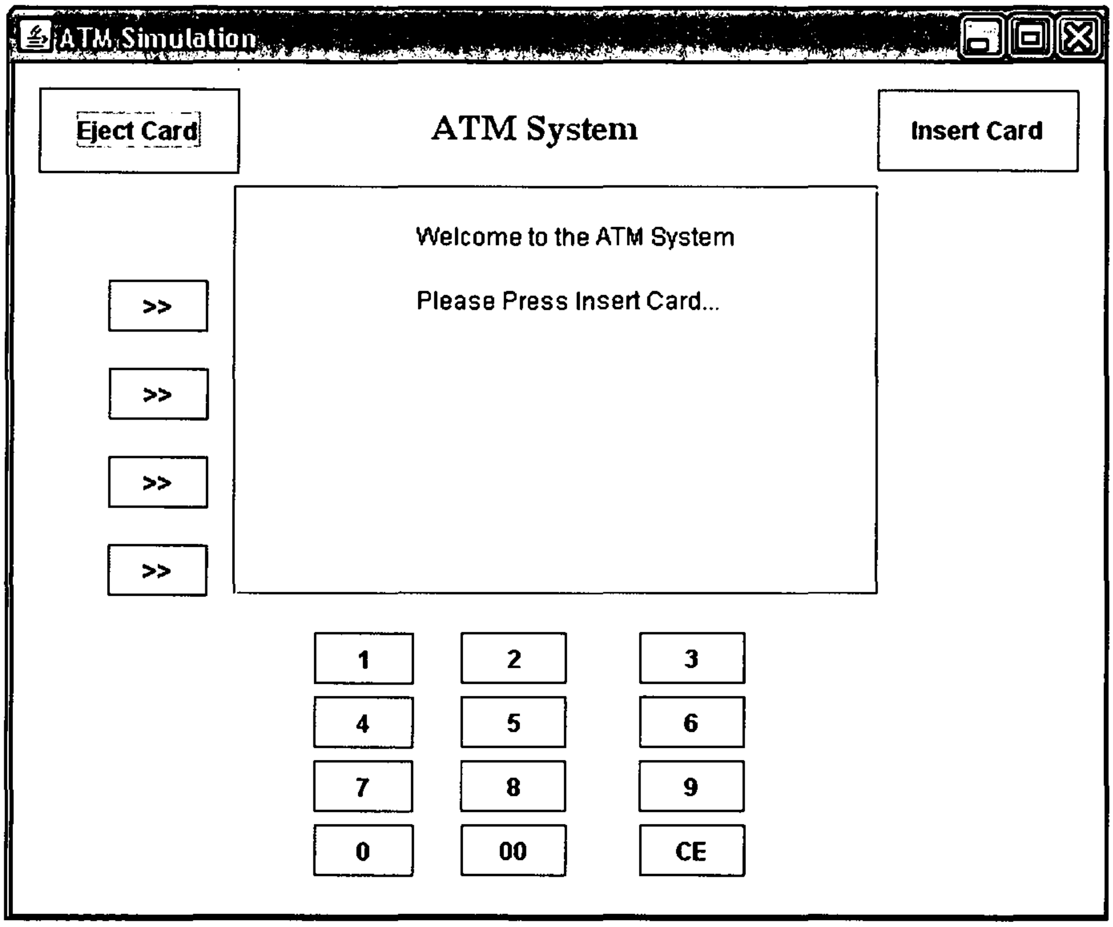
The authorized customer must be able to:

- check their account balance.
- make a cash withdrawal.
- make a transfer of money to any other account liked to the bank.

- abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.

4.6.1 ATM System: The Functional Layer

In our application, we simulate this part of the ATM System by asking the customer to click on Insert Card button (Figure 4-7 shows an example of the screen shots).

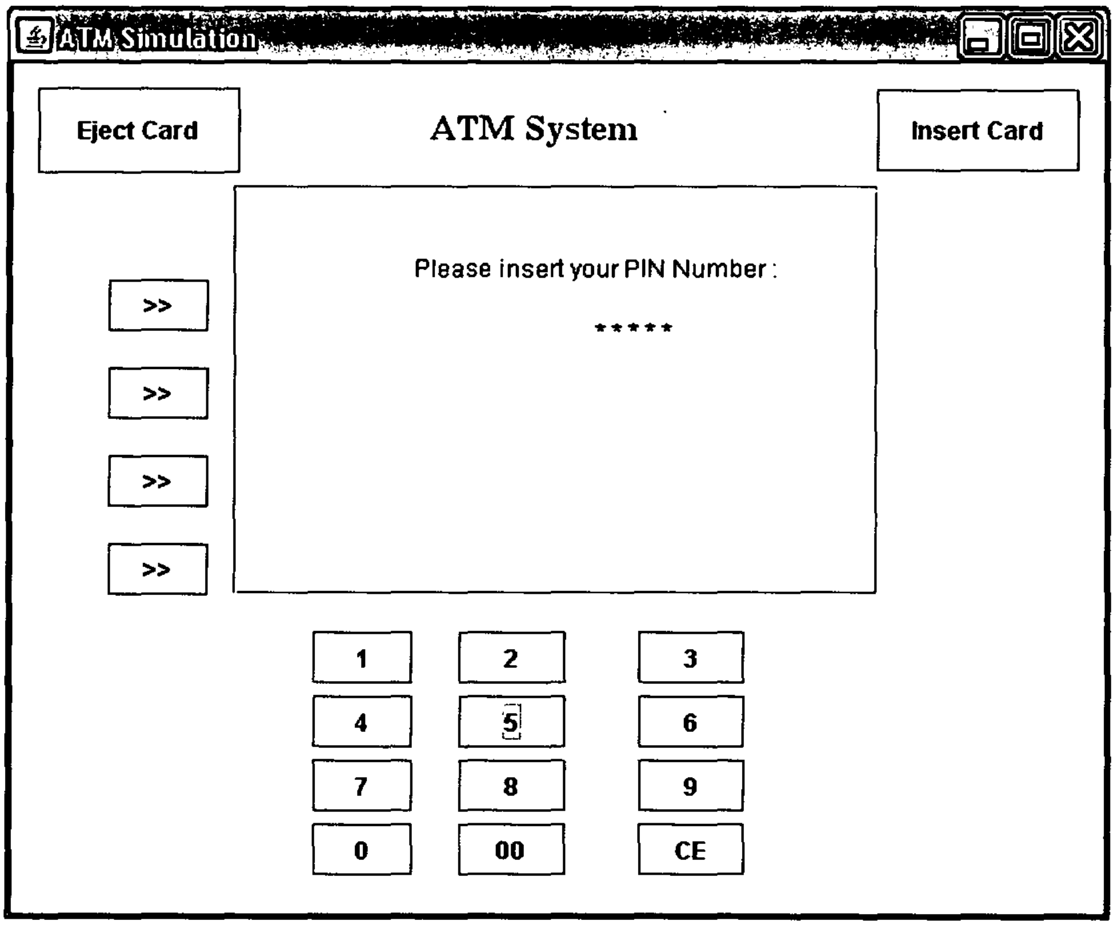


Welcome to the ATM System

Please Press Insert Card...

Figure 4-7: Welcome Screen

In Figure 4-8, as in the ATM at the bank sites, after clicking on Insert Card button, the ATM system will ask the customer to enter PIN code. The customer will enter his/her PIN using the keypad. The PIN consists of six digits. After the last digit has been entered from the keypad, the system accesses the bank database to check this code.



Please insert your PIN Number :

*****

Figure 4-8: Login Screen

If the code is not found in the bank database, the system will ask the customer to try again. If the code is exist in the bank database, then the customer is successfully login to the system and she/he is able to perform the basic financial transactions. Figure 4-9 show the main menu after logging in.

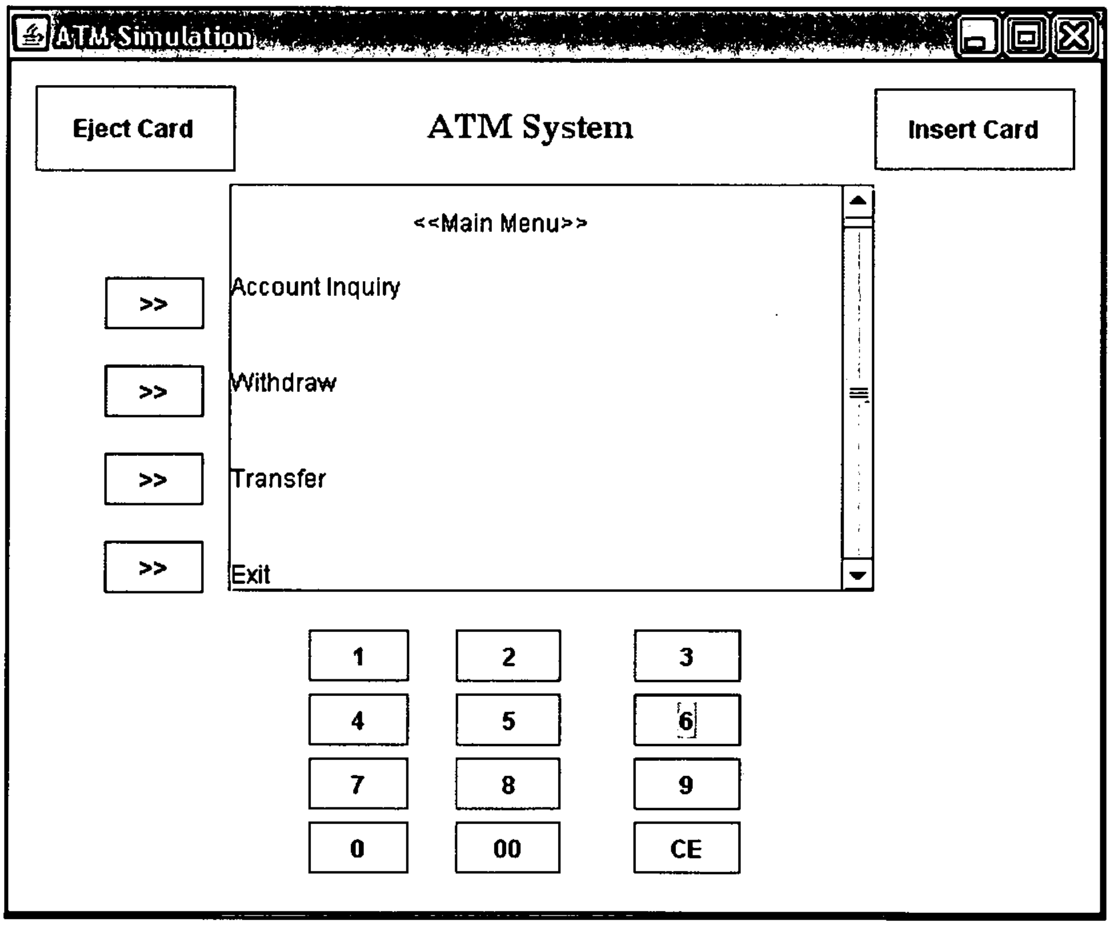
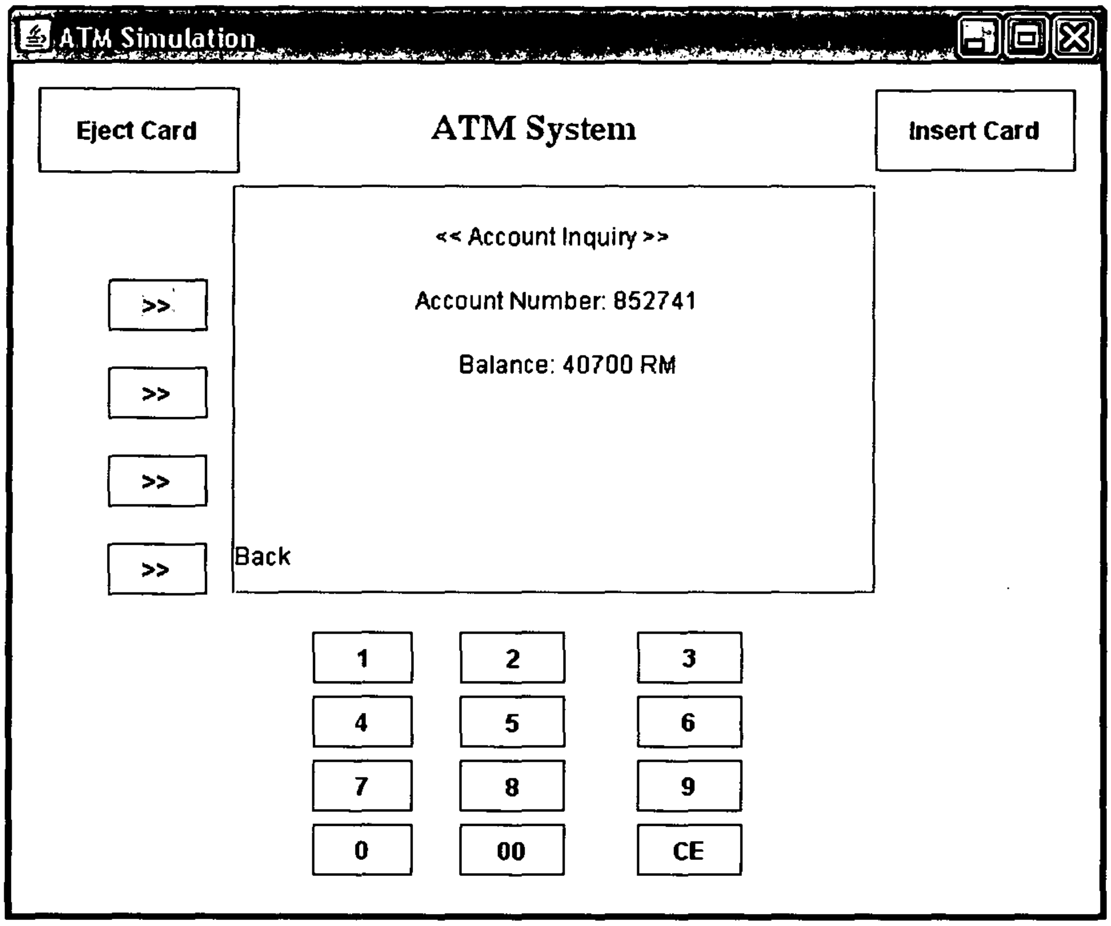


Figure 4-9: ATM System Main Menu

The customer can check her/his account by clicking on the button that near to the text Account Inquiry. Then, the customer can return to the main menu by clicking on Back button (see Figure 4-10).



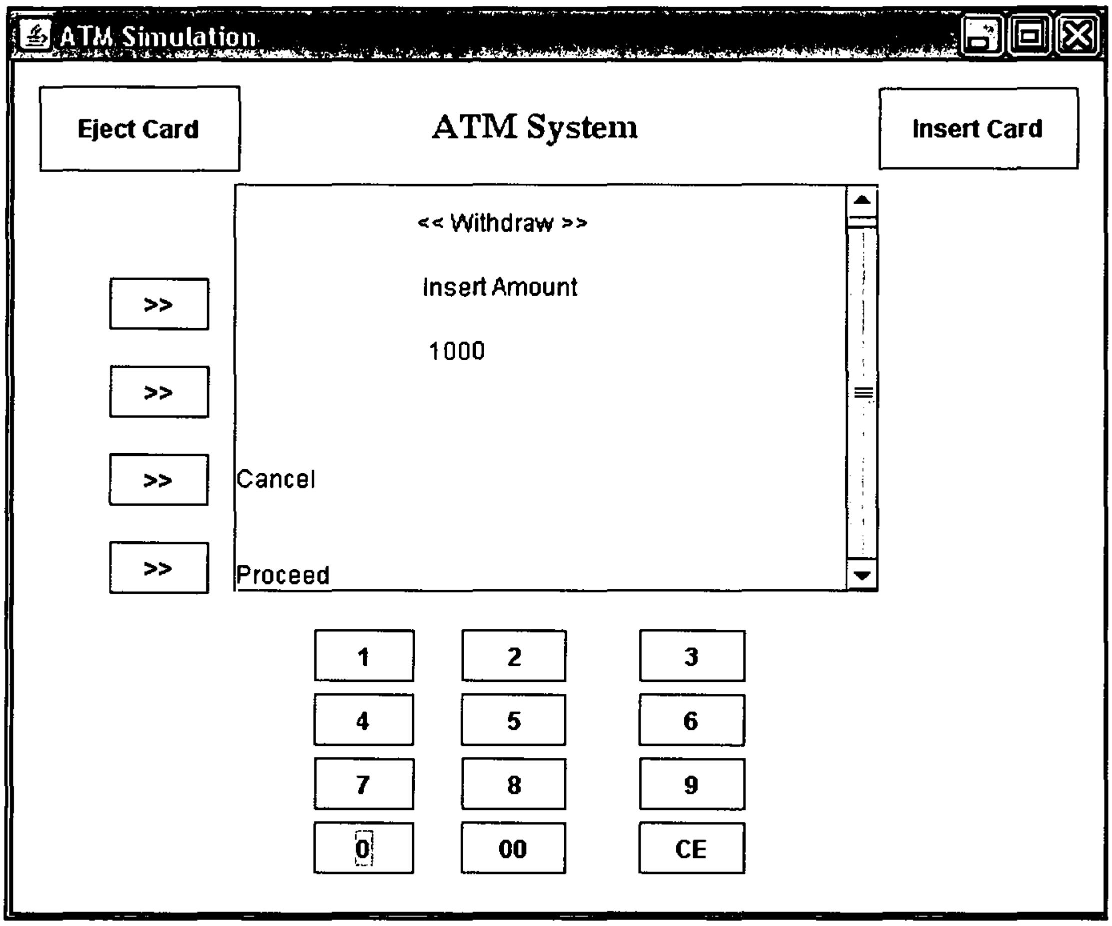
<< Account Inquiry >>

Account Number: 852741

Balance: 40700 RM

Figure 4-10: Account Inquiry

From the main menu, the customer may choose withdrawal option to withdraw cash from the ATM. In this case the customer needs to take his/her cash. In our simulation we just display a report to the customer. See Figure 4-11 and Figure 4-12.



<< Withdraw >>

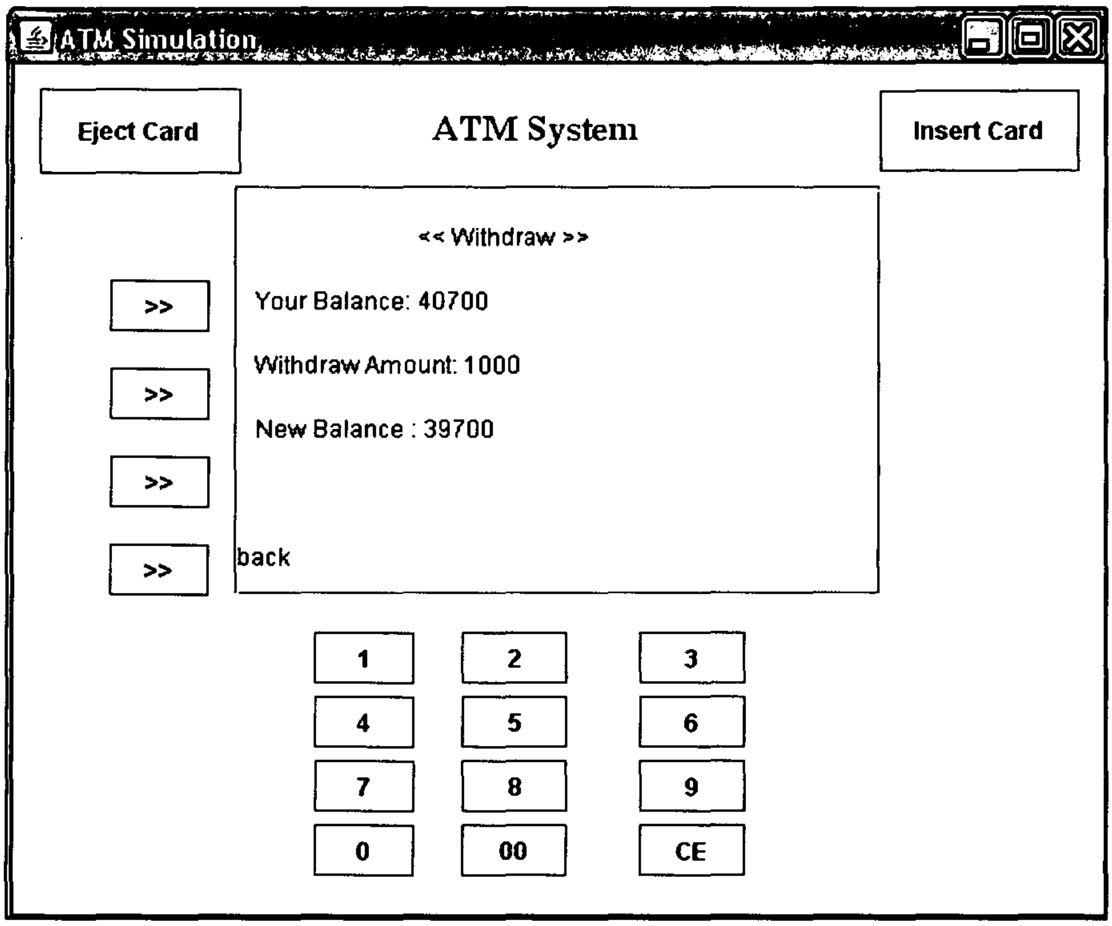
Insert Amount

1000

Cancel

Proceed

Figure 4-11: Withdrawal Money



<< Withdraw >>

Your Balance: 40700

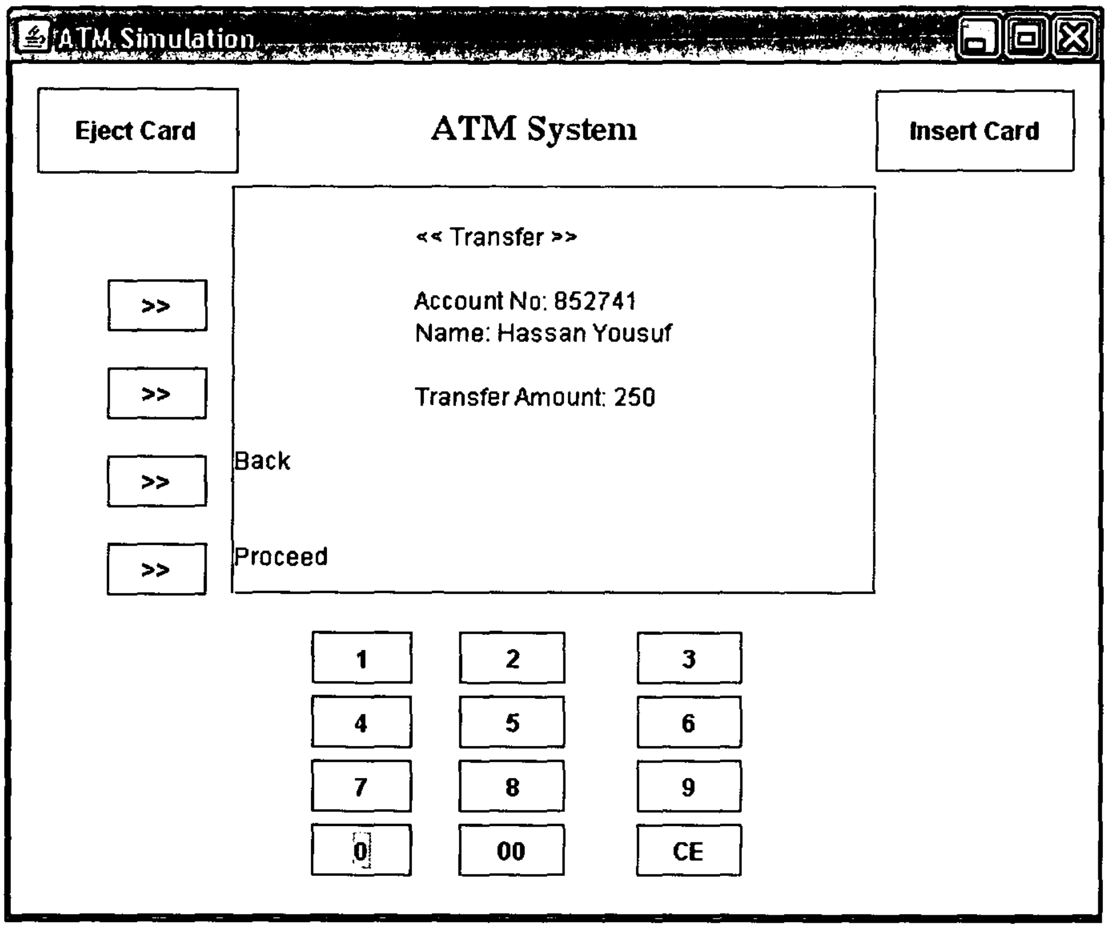
Withdraw Amount: 1000

New Balance : 39700

back

Figure 4-12: Withdrawal Report

Figure 4-14 illustrates the transfer transaction details, the destination account number, the destination name, and the amount.



<< Transfer >>

Account No: 852741
Name: Hassan Yousuf

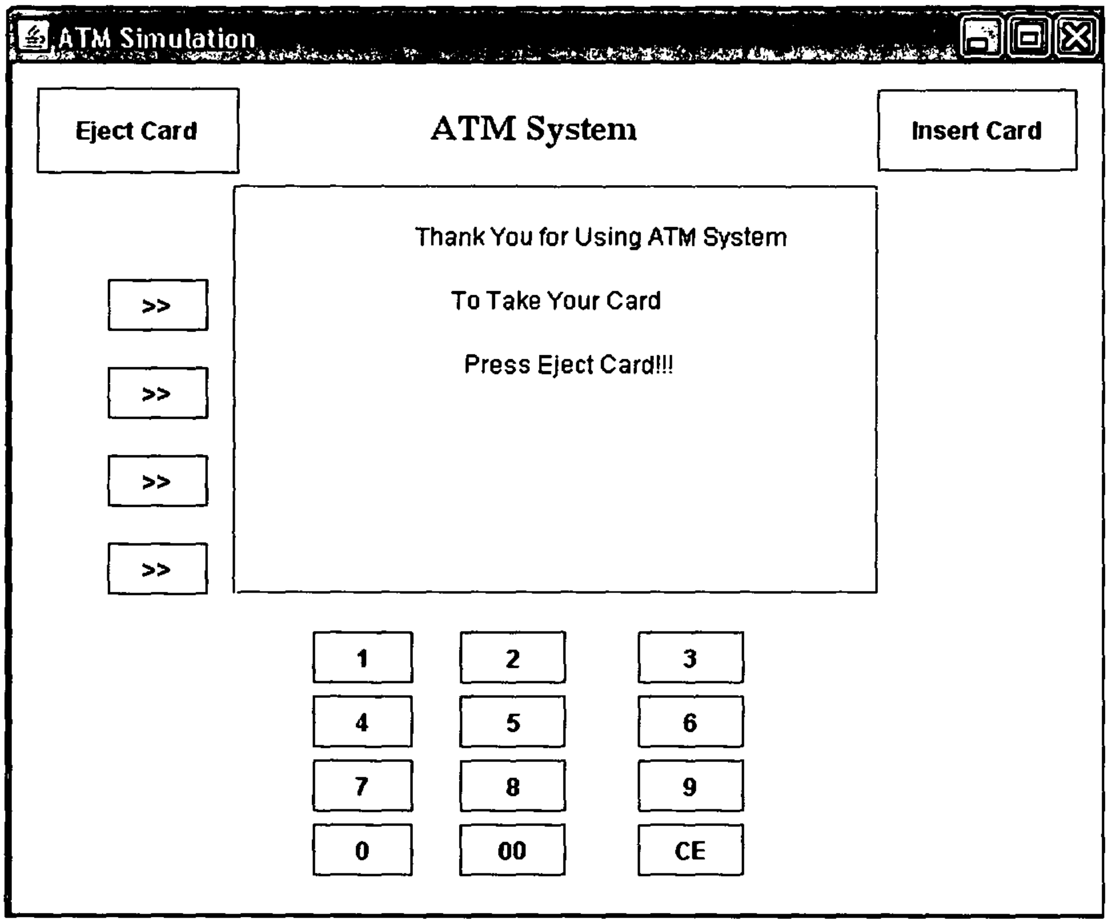
Transfer Amount: 250

Back

Proceed

Figure 4-14: Transfer Transaction: Confirm Operation

When the customer decides to exit from the system, a message will be displayed to inform the customer to take her/his plastic card (see Figure 4-15).



Thank You for Using ATM System

To Take Your Card

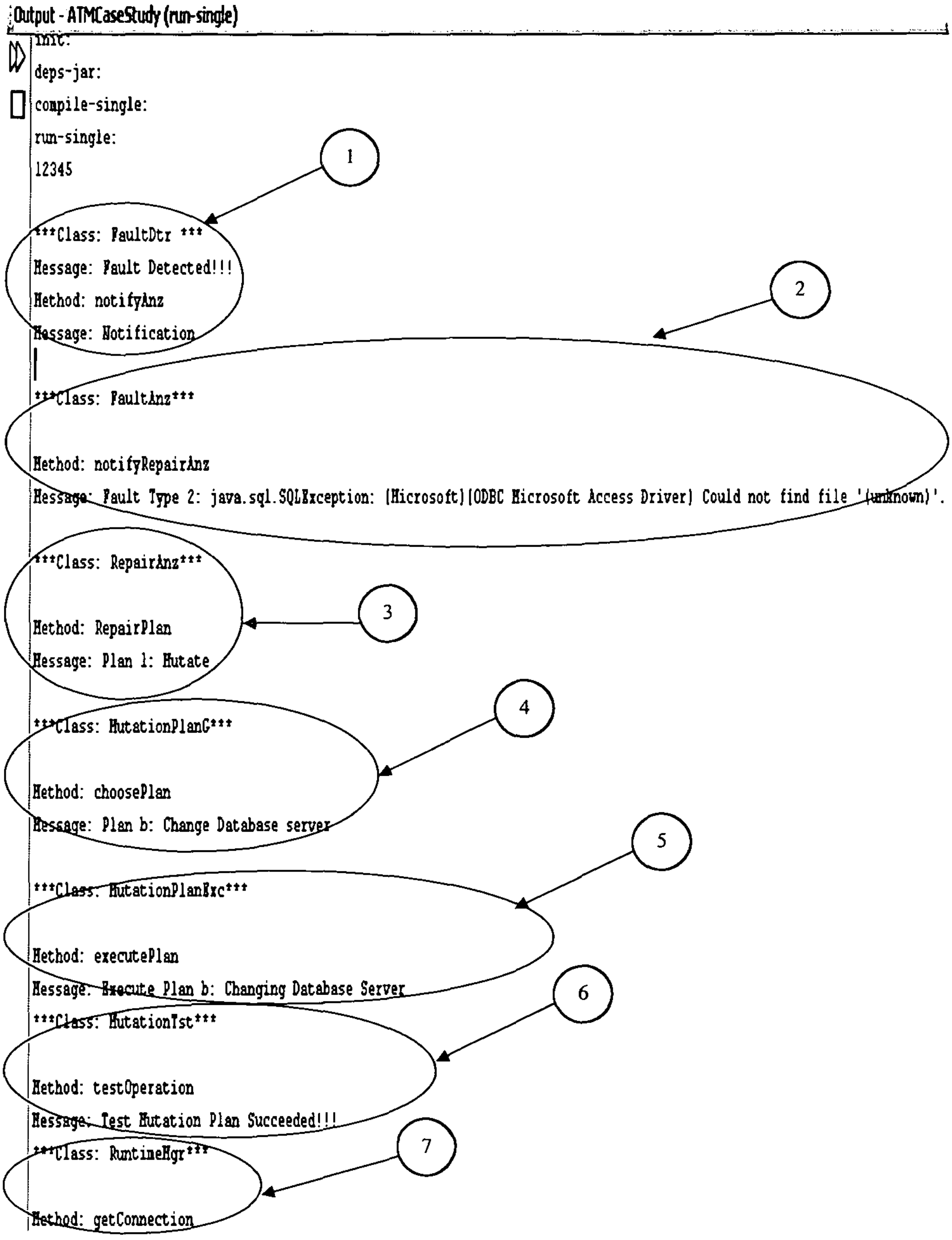
Press Eject Card!!!

Figure 4-15: Exiting From ATM System

4.6.2 ATM System: The Healing Layer

Each time the customer performs a financial transaction using the ATM, the system needs to access the bank database. The database contains all the account information of the customer. In our application, we assume that the server that contains the required information of the bank customer has encountered a failure. Fortunately, a backup from this information is located in another server. This failure might happen during the execution of any transaction.

The failure of the database server which is located in the bank site leads to a failure into the ATM system. If the system fails during a transaction operation, customers will not trust in the services that provided by the ATM system. However, the system must be able to detect the failure of the database server and must be able to access the backup server without the awareness of the customer.



```
init:
deps-jar:
compile-single:
run-single:
12345

***Class: FaultDtr ***
Message: Fault Detected!!!
Method: notifyAnz
Message: Notification

***Class: FaultAnz***

Method: notifyRepairAnz
Message: Fault Type 2: java.sql.SQLException: [Microsoft][ODBC Microsoft Access Driver] Could not find file '(unknown)'.

***Class: RepairAnz***

Method: RepairPlan
Message: Plan 1: Mutate

***Class: MutationPlanC***

Method: choosePlan
Message: Plan b: Change Database server

***Class: MutationPlanExc***

Method: executePlan
Message: Execute Plan b: Changing Database Server
***Class: MutationTst***

Method: testOperation
Message: Test Mutation Plan Succeeded!!!
***Class: RuntimeMgr***

Method: getConnection
```

Figure 4-16: The Output of the Healing Modules in the Healing Layer

4.7 Results and Discussion

The prototype proves that our self-healing software system architecture can be implemented into software application. Using Java language, we implemented each module in the proposed architecture into Java class. Each class performs the task of a specified module.

We considered the Java objects as the system components. The application returns the object to its normal execution after an exception has occurred. The results show that the application has the ability to fix anomalies conditions. The first Java applications can make a copy of the file that is corrupted or deleted. The second application can change the server in order to access the database that it uses.

CHAPTER FIVE: CONCLUSIONS AND FUTURE WORKS

This chapter is divided into two sections. The first section introduces the conclusions of the research. The second section presents the recommendations for future research.

5.1 Conclusions

Autonomic computing is a new area of research which aims to provide software-based systems that have the ability to manage itself at runtime to handle such things as changing user needs, resource variability, changing environment requirements, and systems faults. The major characteristics of autonomic computing are self-configuring, self-healing, self-optimizing, and self-protecting.

Self healing software system is software that has the ability to automatically and continuously monitor, diagnose and adapt itself without human intervention to handle faults that happen during the execution time. Self-healing characteristic has begun to emerge as an interesting, exciting, and potentially valuable property in software systems.

Biological systems have introduced to the world many unforeseen concepts. These systems can handle many challenges with elegance still out of current human artifacts. From this observation, biological inspired software systems approaches have been proposed in the past years in order to handle the complexity of software systems.

This thesis presents software system architecture with self-healing characteristics. The proposed architecture is based on biological system that has the ability to heal by itself (the wound-healing process). The architecture consists of two layers; functional and healing layers. In the functional layer, the system components operate and interact with each other without any disruptions. The healing layer aims to provide the ability for the system to handle anomalous conditions.

Theoretical and formal descriptions of the proposed architecture have been presented in this thesis. The formal description is introduced to prove the functionality of the architecture. This description showed that the architecture has the closure property which indicates that the healed component will have the properties of the original component of the system. Moreover, a Finite State Machine called graphnet has been presented. The graphnet illustrated the component states and the transitions between them during the healing process. The graphnet depicted the healing process for single as well as concurrent faults. The healing process of single-fault is simple process because the system faces one component failure. In contrast, the healing process of concurrent-fault is more complicated. Healing concurrent component failure requires more than one module in each phase of the healing layer.

At the end of this work, a prototype for the proposed architecture has been introduced. The prototype showed that the architecture can be implemented into system application. We provided a Java application. The presented application treats Java object as a system component. If the object failed to provide its service during runtime, the application has the ability to return the object to its normal execution. The presented prototype has been applied into two case studies, simple Java application and ATM system application.

As a conclusion, this thesis presents self-healing software system architecture as well as the specification logics for this architecture. The proposed architecture has the ability to recover the system from single-fault as well as concurrent-faults. The two case studies prove that the proposed architecture can be applied into software system. The fault can occur at any time during the system execution. The healing layer completes the healing process without the awareness of the user. The output from the healing modules in the healing layer shows that each module performs its task and notifies the next modules by sending the required information.

5.2 Future Trends

This work can be pursued further in a number of directions:

- In the case studies, we have simulated the mutation plan for single-fault in which the system can heal from known faults. We left the replication part as well as healing from unknown faults for future works.

- Implementing a prototype for concurrent faults for the proposed architecture. The prototype can be implemented using advance programming languages.

- The proposed architecture consists of two layers, the functional and healing layers. These layers appear as two different systems. As future work, enhancing the proposed architecture is highly recommended. We predict that the functional and healing layer can be composed into one layer.

- A combination of self-healing and self-protecting is highly recommended in order to stop viruses' attacks during the healing process and to heal from a system failure that has been occur after an attack by virus.

- Developing self-healing middleware for large-scale distributed ubiquitous software systems based on the biological systems to heal concurrent-faults in multiple-applications in multiple-platforms. This middleware will increase the reliability of the distributed application and support the interoperability among the system components in a heterogeneous environment.

PUBLICATIONS

1. **Mazin Elhadi**, Azween Abdullah, and Low Tang Jung, "Biologically Inspired Self-Healing Software System Architecture", National Postgraduate Conference, Universiti Teknologi PETRONAS, Tronoh, Malaysia, March 2008.

2. **Mazin Elhadi** and Azween Abdullah, "Layered Biologically Inspired Self-Healing Software System Architecture", Third International Symposium on Information Technology 2008 (ITSim'08), Universiti Kebangsaan Malaysia, Malaysia, August 2008. (accepted for presentation)

REFERENCES

Alan, G., G., & Thomas, A., C. (2003). The dawning of the autonomic computing era. *IBM System Journal, 42,* 5-18.

Aaron, B. B., & Charlie, R. (2005). Measuring the effectiveness of self-Healing autonomic systems. *Proceedings of the Second International Conference on Autonomic Computing (ICAC'05)* (pp. 328-329). IEEE.

Alessandra, G. (2007). *Towards design for self-healing. Presented at the Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting,* Dubrovnik, Croatia: ACM.

Bogdan, S., Dan, L., Marin, L., & Mircea, M. (2007). Towards a real-Time reference architecture for autonomic systems. *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research.* 124-136: ACM.

Daniel, A. M., & Jeffery, O. K. (2007). Guest editors' introduction: Autonomic computing. *IEEE Educational Activities Department, 11*(1), 18-21.

Dashofy, E. M., André, V. D. H., & Richard, N. T. (2002). Towards architecture-based self-healing systems. *Proceedings of the first workshop on Self-healing systems* (pp. 21-26). Charleston, South Carolina: ACM.

David, K., & Heather, O. *The basic principles of wound healing.* Retrieved from http://www.pilonidal.org/pdfs/Principles-of-Wound-Healing.pdf.

David, M., Alla, S., Ian, W., & Steve, R. (2004). Unity: Experiences with a prototype autonomic computing system. *Proceedings of the International Conference on Autonomic Computing (ICAC'04)* (pp. 140-147). IEEE.

Davide, T. (2004). *Research perspective in self-Healing systems.*, University of Milano-Bicocca.

Deepak, K. G. (2005). Meta dynamic states for self healing autonomic computing systems. *International Conference on Man and Cybernetics Systems* (Vol. 1, pp. 39-46). IEEE.

Dewayne, E. P., & Alexander, L. W. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes, 17*(4), 40-52. ACM.

Eleni, P., & Nancy, A. (2006). A framework for the deployment of self-managing and self-configuring components in autonomic environments. *Proceedings of the 2006 International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM'06)* (p. 5). IEEE.

Fabio, K., Fabio, C., Gordon, B., & Roy, H. C. (2002). The case for reflective middleware. *Communications of the ACM, 45*(6), 33-38. ACM.

Fuad, M. M. (2007). *AN aUTONOMIC sOFTWARE aRCHITECTURE.* Unpublished doctoral dissertation, MONTANA STATE UNIVERSITY, Computer Science (1, Vol. 1).

George, C., Vincent, H. B., Ian, D. G.-D., & Chad, B. (2006). Practical autonomic computing. *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)* (pp. 3-14). IEEE Computer Society.

Grishikashvili, Pereira. E, & Pereira. R. (2007). Simulation of fault monitoring and detection of distributed services. *Simulation Modelling Practice and Theory* (4, Vol. 15, pp. 492-502). Elsevier.

Haydarlou, A., R, Overeinder, B., J, & Brazier, F., M. (2005). A self-healing approach for object-oriented applications. *Proceedings of the 3rd International Workshop on Self-Adaptive and Autonomic Computing Systems* (pp. 191-195). IEEE.

IBM Research. (2001). *Autonomic computing.* Retrieved from Http://www.research.ibm.com/autonomic/ overview.

Jayne, C., & Sarah, K. (2004). The wound healing process. *Peterborough Wound Care.*

Jeffery, O., Kephart, & David, M., Chess. (2003). The vision of autonomic computing. *IEEE Computer Society Press, 36,* 41-50.

Jochen, W. (2007). An approach to detecting failures automatically. *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting,* (pp. 17-24). Dubrovnik, Croatia: ACM.

King, T. M., Babich, D., Alava, J., Clarke, P. J., & Stevens, R. (2007). Towards self-Testing in autonomic computing systems. *Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07)* (pp. 51-58). IEEE.

Laddaga, R. (1999). Creating robust software through self-adaptation. *IEEE Intelligent Systems, 14*(3), 26-29. IEEE.

Manish, P., & Salim, H. (2005). Autonomic computing: An overview. *Lecture Notes in Computer Science, Unconventional Programming Paradigms, 3566,* 257-269. SpringerLink.

Marija, M.-R., Nikunj, M., & Nenad, M. (2002). Architectural style requirements for self-Healing systems. *Proceedings of the first workshop on Self-healing systems* (pp. 49-54). Charleston, South Carolina: ACM.

Mary, S., & David, G. (1996). *Software architecture: Perspectives on an emerging discipline.* Prentice-Hall.

Mazeiar, S., & Ladan, T. (2005). Autonomic computing: Emerging trends and open problems. *ACM_SIGSOFT Software Engineering Notes, 30*(4), 1-7. ACM.

Michael, E. S. (2004). Self-Healing components in robust software architecture for concurrent and distributed systems. *Science of Computer Programming. 57*(1), 27-44. Elsevier.

Michael, E. S., & Daniel, C. (2005). Connector-Based self-Healing mechanism for components of a reliable system. *ACM_SIGSOFT Software Engineering Notes, 30*(4), 1-7. ACM.

Michael, J., Jing, Z., David, R., & John, S. (2007). *A modeling framework for self-healing software systems.* (Autonomics Research, Motorola Network Infrastructure Research Lab).

Mohammad, R., Nami, & Koen, B. (2007). A survey of autonomic computing systems. *Proceedings of the Third International Conference on Autonomic and Autonomous Systems* (p. 26). IEEE Computer Society.

Mohammad, R., Nami, & Mohsen, S. (2007). Autonomic computing: A new approach. *First Asia International Conference on Modelling & Simulation (AMS '07)* (pp. 352-357). IEEE.

Paskorn, C., & Junichi, S. (2006). A biologically-inspired autonomic architecture for self-healing data centers. *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)* (Vol. 1, pp. 103-112). IEEE Computer Society.

Paul, H. (2001). Autonomic computing: IBM's perspective on the state of information technology. *IBM Corporation.*

Paul, L., Alexander, M., & John, L. (2005). Defining autonomic computing: A software engineering perspective. *Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)* (pp. 88-97). IEEE.

Pruet, B., & Junichi, S. (2002). BiSNET: A biologically-inspired middleware architecture for self-managing wireless sensor networks. *Computer Networks*. *51*(16), 4599-4616. Elsevier.

Robert, F., Diegelmann, & Melissa, C., Evans. (2004). Wound healing: An overview of acute, fibrotic and delayed healing. *Frontiers in Bioscience,* pp. 283-289.

Rog'erio, d., Lemos, Jon, T., Modupe, A., & Simon, F. (2007). Immune-Inspired adaptable error detection for automated teller machines. *IEEE Transactions on Systems, Man, and Cybernetics-Part C: Applications And Reviews* (Vol. 37, pp. 873 - 886). IEEE.

Sarah, C. (2002). Wounds the healing process. *Royal Pharmaceutical Society of Great Britain.*

Sasitharan, B., Dmitri, B., William, D., Mícheál.Ó F., & John, S. (2006). Biologically inspired self-governance and self-organisation for autonomic networks.

Proceedings of the 1st international conference on Bio inspired models of network, information and computing systems. Cavalese, Italy: ACM.

Selvin, G., David, E., & Lance, D. (2002). A biologically inspired programming model for self-healing systems. *Proceedings of the first workshop on Self-healing systems* (pp. 102-104). Charleston, South Carolina: ACM.

Selvin, G., David, E., & Steven, M. (2003). A biological programming model for self-Healing. *Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems: In association with 10th ACM Conference on Computer and Communications Security* (pp. 72-81). Fairfax, VA: ACM.

Shameem, A., Sheikh, I. A., Moushumi, S., & Munirul, M. H. (2007). Self-healing for autonomic pervasive computing. *Proceedings of the 2007 ACM symposium on Applied computing* (pp. 110-111). Seoul, Korea: ACM.

Shang-Wen, C., David, G., Bradley, R. S., João, P., Sousa, Bridget, S., Peter, S. et al. (2002). Software architecture-Based adaptation for pervasive systems. *Proceedings of the International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing, 2299,* 67-82. Springer-Verlag.

Shin, M. E., & Jung, H. A. (2006). Self-Reconfiguration in self-Healing systems. *Proceedings of the Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems* (pp. 89-98). IEEE Computer Society.

Stuart, A., Mark, H., Rob, P., Mark, R., Roger, S., James, S. et al. (2003). Making autonomic computing systems accountable: The problem of human-Computer interaction. *Proceedings of the 14th International Workshop on Database and Expert Systems Applications (DEXA '03)* (pp. 718-724). IEEE.

Wang, C., Li, Y., & Bu, J. (2004). *A biological formal architecture of self-healing system.* Presented at the 2004 IEEE International Conference on Systems, Man and Cybernetics. IEEE.

WoundHeal. Retrieved 22, Nov, 2007, from <http://www.woundheal.com/healing/processIndex.htm>.

Yang, Q., Yang, X.-C., & Xu, M.-W. (2005). A framework for dynamic software architecture-based self-healing. *ACM SIGSOFT Software Engineering Notes, 30*(4), 1-4. ACM.

Zach, P., & Sam, S. (2005). A machine to support autonomic computing*2005 IEEE Region 5 and IEEE Denver Section Technical, Professional and Student Development Workshop* (pp. 25-31). IEEE.

Zeid, A., & Gurguis, S. (2005). Towards autonomic web services. *The Fourth International Conference on Computer and Information Technology, CIT '04* (p. 69). IEEE.

APPENDIX A: TEST PROGRAM LISTING

APPENDIX A.1: Test Program 1

```
package RunSimulation;


import java.io.*;
public class FileAccess {
  RandomAccessFile f;
  public FileAccess(String fileName, String name, String pass) {
    System.out.print("***File Access Class***\n");
   openFile(fileName, name, pass);
  }

  public void openFile (String fileName, String name, String pass) {
    System.out.print("Function: nopenFile\n");
      try {
        System.out.print("\n++++++++file name is  "+fileName+"\n");
        f = new RandomAccessFile(fileName, "rw");
        setData(name, pass, f);
      } catch (FileNotFoundException ex) {
        System.out.print("File "+fileName+" Not Found!!!\n"+ ex);
        new FaultDtr(ex);
      }
  }

  public void setData(String name, String pass, RandomAccessFile f){
    System.out.print("Function setData\n");
    try {
      f.writeUTF(name);
```

APPENDIX A.1: Test Program 1 (continued)

```
        f.writeUTF(pass);
      } catch (IOException ex) {
        System.out.print("Input Output Exception!!!\n"+ ex +"\n");
        new FaultDtr(ex);
      }
   }
}
```

APPENDIX A.2: Test Program 2

```
package RunSimulation;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;

public class OutputFromFile {
  RandomAccessFile f;
  public OutputFromFile(String fileName){
     try {
          f = new RandomAccessFile(fileName, "r");
          System.out.print("Name: "+ getName() + "\n");
          System.out.print("Password: " + getPass() + "\n");

        } catch (FileNotFoundException ex) {
          System.out.print("File not Found!!\n" + ex);
          new FaultDtr(ex);
        }

  }
  public String getName(){
    String str = "";
    try {
       str = f.readUTF();
    } catch (IOException ex) {
      System.out.print("Input Output Exception!!\n"+ex);
      new FaultDtr(ex);
    }
                              return str;
```

APPENDIX A.2: Test Program 2 (continued)

```
}
  public String getPass(){
    String str = "";
    try {
      str = f.readUTF();
    } catch (IOException ex) {
      System.out.print("Input Output Exception!!\n"+ex);
      new FaultDtr(ex);
    }
    return str;
  }

}
```

APPENDIX A.3: Test Program 3

```
package RunSimulation;


public class FileLocations {
   public static final String loc1 = "c://Data.txt";
   public static final String loc2 = "c://backup//Data.txt";
   public FileLocations(){


   }
   public static String getFile(int loc){
      String str = "";
      switch (loc){
         case 1:
            str = loc1;
            break;
         case 2:
            str = loc2;
      }
      return str;
   }
}
```

APPENDIX A.4: Test Program 4

```
package RunSimulation;

import java.io.*;

public class FaultDtr {
  public FaultDtr(Exception ex){
    System.out.print("\n***Class: FaultDtr ***\nMessage: Fault Detected!!!\n");
    notifyAnz(ex);
  }
  public void notifyAnz(Exception ex){
    System.out.print("Method: notifyAnz\nMessage: Notification\n");
    new FaultAnz(ex);
  }
  public void notifyFaultRst(){

  }

}
```

APPENDIX A.5: Test Program 5

```
package RunSimulation;


import java.io.*;


public class FaultAnz {
   public FaultAnz(Exception ex){
      System.out.print("\n***Class: FaultAnz***\n");
      notifyRepairAnz(ex);
   }
   public void notifyRepairAnz(Exception ex){
      System.out.print("\nMethod: notifyRepairAnz\nMessage: ");
      if (ex instanceof FileNotFoundException)
      {
         System.out.print("Fault Type 1: "+ex+"\n");
         new RepairAnz(1);
      }
      else if (ex instanceof IOException){
         System.out.print("Fault Type 2: "+ex+"\n");
         new RepairAnz(2);
      }
      else{
         System.out.print("Fault Type -1: "+ex+"\n");
         new RepairAnz(-1);
      }
   }
}
```

APPENDIX A.6: Test Program 6

```
package RunSimulation;

public class RepairAnz {
   public RepairAnz(int faultID){
      System.out.print("\n***Class: RepairAnz***\n");
      repairPlan(faultID);
   }
   public void repairPlan(int faultID){
      System.out.print("\nMethod: RepairPlan\nMessage: ");
      switch (faultID){
         case 1:
         case 2:
            System.out.print("Plan 1\n");
            new MutationPlanG(faultID);
            break;
         case 3:
            System.out.print("Plan 2\n");
            new ReplicateExc();
         default:
            System.out.print("Plan 3\n");
      }
   }
}
```

APPENDIX A.7: Test Program 7

```
package RunSimulation;

class MutationPlanG {

  public MutationPlanG(int faultID) {
    System.out.print("\n***Class: MutationPlanG***\n");
    choosePlan(faultID);
  }
  public void choosePlan(int faultID){
    System.out.print("\nMethod: choosePlan\nMessage: ");
    switch (faultID){
      case 1:
        System.out.print("Plan a\n");
        new MutationPlanExc('a');
        break;
      case 2:
        System.out.print("Plan b\n");
        new MutationPlanExc('b');
        break;
      default:
        System.out.print("Plan z\n");
        new MutationPlanExc('z');
    }
  }

}
```

APPENDIX A.8: Test Program 8

```
package RunSimulation;

class MutationPlanExc {

  public MutationPlanExc(char c) {
    System.out.print("\n***Class: MutationPlanExc***\n");
    executePlan(c);
  }
  public void executePlan(char c){
    System.out.print("\nMethod: executePlan\nMessage: ");
    switch (c){
      case 'a':
        System.out.print("Execute Plan a\n");
        new CopyFile(1);
        break;
      case 'b':
        System.out.print("Execute Plan b\n");
        //new planB();
        break;
      default:
        System.out.print("Execute Plan z\n");
        //new planC();
    }
  }


}
```

APPENDIX A.9: Test Program 9

```
package RunSimulation;
import java.io.*;
import java.util.logging.*;

public class CopyFile {
   public CopyFile(){
     System.out.print("Class: CopyFile\n");
   }

   CopyFile(int i) {
      System.out.print("Class: CopyFile\n");
      openFile(i);
   }
   public void openFile(int i){
      try {
         System.out.print("Method: openFileData\n");
         RandomAccessFile newFile = new RandomAccessFile(FileLocations.getFile(i),
"rw");
         RandomAccessFile backupFile = new
RandomAccessFile(FileLocations.getFile(2), "r");
         copyFileData(newFile, backupFile);
      } catch (FileNotFoundException ex) {
         System.out.print("Message: File Not Found!!!\n"+ex);
      }
   }
   public void copyFileData(RandomAccessFile newFile, RandomAccessFile
backupFile){
      try {
         System.out.print("Method: copyFileData\n");
```

APPENDIX A.9: Test Program 9 (continued)

```
            String str;
            str = backupFile.readUTF();
            while (str != null) {
              System.out.print("str = "+ str + "\n");
              newFile.writeUTF(str);
              str = backupFile.readUTF();
            }
            newFile.close();
            backupFile.close();
            System.out.print("\nMessage: File Copied!!!\n");
          } catch (IOException ex) {
            System.out.print("Message: Input Output Exception!!!\n"+ex);
          }
      }
  }
```