SELF-RECONFIGURATION
IN SELF-HEALING SYSTEMS

by

AN, JUNG HOON

A THESIS

IN

COMPUTER SCIENCE

Submitted to the Graduate Faculty of
Texas Tech University in
Partial Fulfillment of the
Requirements for the
Degree of

MASTER OF SCIENCE

Approved

Michael (Eonsuk) Shin

Chairperson of the Committee

Yu Zhuang

Fred Hartmeister

Dean of the Graduate School

December, 2008

ACKNOWLEDGEMENTS

I would like to show my appreciation for my thesis advisor, Dr. Michael E. Shin for great patience and support and express appreciation to my wife and family for their endless love and support. Moreover, I am grateful to all my friends, and colleagues who encourage me.

I thank Texas Tech University, the United States Government, and the Korean government for giving me a wonderful opportunity to study and change my life.

Finally, I would like to thank the holy name of JESUS who gives me wonderful love everyday.

ABSTRACT

The purpose of this thesis is to suggest an approach to self-reconfiguration which is part of a self-healing mechanism against anomalous objects of a system prior to repairing anomalies of objects. The approach assumes that the software architecture of a system is structured into components and connectors between the components. A 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. The elevator system with multiple elevators is considered for a case study of self-reconfiguration.

TABLE OF CONTENTS

Acknowledgement.....	ii
Abstract.....	iii
List of figures.....	vi
I . Introduction.....	1
II . Related work.....	3
2.1. Introduction	3
2.2. Reconfiguration approaches	3
2.2.1. Jeff Kramer and Jeff Magee’s Approach	3
2.2.2. Dr. Hassan Gomaa’s Approach.....	4
2.2.3. João Paulo A. Almeida, Maarten Wegdam, Luís Ferreira Pires, and Marten van Sinceren’s Approach.....	4
2.2.4.Jasminka Matevska and Wilhelm Hasselbring’s Approach.....	5
2.2.5. Dong Kwan Kim and Shawn Bohner’s Approach.....	5
2.2.6. Riadh Ben Halima, Khalil Drira and Mohamed Jmaiel’s Approach	6
2.2.7. Xabier Elkorobarrutia, Mikel Muxika, Goiuria Sagardui’s Approach	6
2.3. Summary.....	7
III . Problem statement and research approach.....	8
3.1. Introduction	8
3.2. Problem Statement	8
3.3. Research Approach.....	9
3.4. Summary.....	9
IV . Reconfiguration mechanism	10
4.1. Self-Healing Systems.....	10
4.1.1. Software Architecture for Self-Healing Systems	10
4.1.2. Self-healing Component Architecture	11
4.1.3. Self-Healing Connector Architecture	14

4.2. Anomaly Detection in Self-Healing Systems	16
4.3. Approach to Dynamic Reconfiguration in Self-Healing Systems	17
4.4. Reconfiguration of Objects in a Component.....	18
4.4.1. Anomalous Tasks	19
4.4.2. Anomalous Passive Object Accessed by Tasks	22
4.4.3. Anomalous Connectors between Tasks	25
4.4.4. Impact of Reconfiguration of Component.....	26
4.5. Validation	26
4.6. Conclusions	27
V. Validation – case study	28
5.1. Anomaly Detection	28
5.2. Reconfiguration.....	36
5.2.1. Introduction	36
5.2.2. Reconfiguration before the repair.....	37
5.3. Repair	49
5.4. Reconfiguration after repair.....	50
5.5 Conclusions	51
VI. Conclusion	52
References	53

LIST OF FIGURES

4.1: Overview of Software Architecture for Self-Healing Systems	11
4.2: Self-Healing Component Architecture	13
4.3: Self-Healing Connector Architecture	15
4.4: Dynamic self-configuration in Component1	21
4.5: Message Queue Self-Healing Connector	21
4.6: Attributes and Operations of Task2.....	23
4.7: Specification of Task2	24
5.1 : An example of anomaly detection in a component	29
5.2: Statechart for task1 in figure 5. 1	29
5.3: Task architecture of elevator system with connectors.....	31
5.4: Extended task architecture of elevator system with connectors.....	32
5.5: Detailed design of Component monitor.....	33
5.6: Statechart for Elevator manager task	34
5.7: Dynamic self-reconfiguration in Elevator component1	38
5.8: Dynamic reconfiguration for Elevator manager task	39
5.9: Attribute and Operation of Queue2.....	39
5.10: Dynamic reconfiguration for Local elevator status & plan	41
5.11: Specification of Elevator Manager.....	41
5.12: Attributes and Operation of Elevator manager	42
5.13: Dynamic reconfiguration for Queue2 in Elevator component.....	43
5.14: Dynamic reconfiguration for Scheduler component.....	44
5.15: Dynamic reconfiguration for Elevator status & plan server task.....	45

5.16: Dynamic reconfiguration for Local elevator status & plan	46
5.17: Dynamic reconfiguration for Queue3 in Scheduler component	47
5.18: Dynamic reconfiguration for inter components.....	49
5.19: Dynamic reconfiguration after repair for Elevator1 component	50

CHAPTER I

INTRODUCTION

A self-healing system needs to be dynamically self-reconfigured before repairing anomalous objects detected in the system. Self-reconfiguration in a self-healing system isolates anomalous objects from other healthy objects so that the anomalous objects are localized to minimize the interference of healthy objects in repairing the anomalies.

The software architecture for self-healing systems [1, 2, 3] provides a basis for specifying a self-healing mechanism including self-configuration. The detection, reconfiguration, and repair of anomalies in a self-healing system can be affected by the software architecture, in particular, the structure of components and the style of interaction between the components, which is capable of encapsulating a self-healing mechanism. Anomalies in the system may be detected in the objects within components and in the interaction between the components. A self-healing mechanism needs to contain an approach for dynamically reconfiguring components and interaction between the components in a system against anomalies of objects.

This paper describes an approach to self-reconfiguration of a system, which is structured into components and connectors between the components. Anomalous objects in components and connectors between the components are isolated by the approach prior to repair of anomalies of the objects.

This paper is organized as follows. It begins with related works in section 2.

Section 3 describes the problem statement and research approach. Section 4 describes an architectural style, connector-based software architecture for a self-healing system and introduces a self-reconfiguration approach in components. Section 5 describes the validation of self-reconfiguration in the elevator system. Finally section 6 concludes this paper.

CHAPTER II

RELATED WORK

2.1. Introduction

This chapter describes the several approaches to reconfiguration in systems. The chapter is organized as follows: Section 2.2 concentrates on approaches for reconfiguration of systems. Section 2.3 summarizes this chapter.

2.2. Reconfiguration approaches

2.2.1. Jeff Kramer and Jeff Magee's Approach

This approach suggests a dynamic change management model to separate structural concerns from component application concerns [4]. A software system is structured into structural configuration concerns, which specify the change of software architecture, and functional application concerns, which provide a capability of the change to structural configuration concerns. In this approach, a component in the system can change only when it has reached a “quiescent” state, which means the component is isolated completely from other components.

This approach uses a change protocol for establishing independent transactions between structural concerns and application concerns. The change rule includes nodes creation, deletion, connection, and which nodes should be passive and active. Especially, passive and active node change can suggest a concise way to transit from

one state to another state. This distinction guarantees that, for a node to be changed, it should wait for the associated other nodes to reach passive states. By this change protocol, this approach accomplishes a clear manner for dividing the concerns into structural management and application.

2.2.2. Dr. Hassan Gomaa's Approach

This approach [5] is an extension to Jeff Kramer and Jeff Magee's Approach by adding additional states related to the dynamic configuration of components in the software system. This approach describes a dynamic software reconfiguration pattern for real-time software systems which is structured into components [5]. The paper also shows how to apply the patterns to accomplish the reconfiguration while the system is running. This paper proposes an approach to using the software reconfiguration patterns in reconfigurable evolutionary product line life cycle (REPLLC) [5] to execute reconfiguration at runtime environment.

2.2.3. João Paulo A. Almeida, Maarten Wegdam, Luís Ferreira Pires, and Marten van Sinceren's Approach

This approach [6] introduces a dynamic reconfiguration of distributed systems based on object-middleware as well as a dynamic reconfiguration within CORBA. Operations for a dynamic reconfiguration such as replacements, migration, addition, and removal are used to manipulate entities, which can be a component, an object, or multiple numbers of components. A trend of distributed systems has been collaborated

with object-middleware such as CORBA, Java RMI, and DCOM. This object-middleware provides distribution transparency and flexibility for application designers.

The dynamic reconfiguration may be supported at the level of middleware, instead of at each application system level. This approach [6] has introduced an approach to dynamic reconfiguration using middleware. In this approach, an object should be in a safe state before reconfiguring an object, which is similar to the quiescent state. This approach guarantees mutual consistency among components that can be achieved through reaching safe state within limited time.

2.2.4.Jasminka Matevska and Wilhelm Hasselbring's Approach

This approach is to increase service availability during the runtime reconfiguration of a component-based system using the usage model of the system and the component protocol information. The aim of this approach is to provide optimal availability of requested services during reconfiguration for evolution to meet changing requirements or improve quality properties. By considering varying number of requests for a service at different times and the priority of each service, this approach computes the minimal dependency graphs among component instances and the maximal possible availability of services provided by the system.

2.2.5. Dong Kwan Kim and Shawn Bohner's Approach

This approach proposes the dynamic reconfiguration mechanism and its supporting framework for Java application system using Aspect-oriented programming.

This approach can be applied to software systems that are adapted to evolutionary computing needs and environments. To change a software component, the replacing component should reach a quiescent state and the state of the component is stored so as to transfer it to the replaced component. This approach uses aspect-oriented programming technique for the design and implementation of the proposed dynamic reconfiguration mechanism and framework.

2.2.6. Riadh Ben Halima, Khalil Drira and Mohamed Jmaiel's approach

This approach describes QoS management for self-healing Web services [7]. This approach suggests a connector-based healing architecture, which performs monitoring, diagnosis, and repair to heal a system. This approach concentrates on the communication level repair in which a system consists of components and their connectors. The connectors between components are involved in intercepting and adding QoS-related headers so as to send and receive messages. The connectors between components checks whether agreed-upon QoS has been achieved. The repair-level connectors execute the reconfiguration of a system if expected QoS is not satisfied.

2.2.7. Xabier Elkorobarrutia, Mikel Muxika, Goiuria Sagardui's Approach

This approach describes a reconfiguration mechanism for statechart-based software components and a framework supporting the reconfiguration mechanism. A

system consists of components, each of which is modeled using a statechart. A system is reconfigured by reflecting the states of components if the component needs to be changed. To support this reflective architecture, the FraC framework has been described.

2.3. Summary

Most of the approaches to reconfiguration have focused on evolution or expected changing environment, under which a reconfiguration can be planned and executed. However, less attention has been paid to the reconfiguration that is required for self-healing of systems.

CHAPTER III

PROBLEM STATEMENT AND RESEARCH APPROACH

3.1. Introduction

This research deals with an approach to self-reconfiguration, which is part of a self-healing mechanism against anomalous objects. A self-healing system needs to be self-reconfigured at runtime before repairing anomalous objects detected in the system. To do this, a self-reconfiguration mechanism should be developed for self-healing systems. This search describes the self-reconfiguration mechanism and its validation through case studies.

This chapter describes the problem statement for this research and provides the research approach to solve the problem. Section 3.2 addresses the problem statement and Section 3.3 describes the research approach. Section 3.4 gives the summary of the chapter.

3.2. Problem Statement

Current approaches to the reconfiguration of (object-oriented) systems focused on either evolving the systems or making the systems more reliable. However, less attention has been paid to self-reconfiguration for self-healing of systems. It is necessary for developing an approach to self-reconfiguration such that anomalous objects in the systems should be isolated from other healthy objects enough to be treated.

3.3. Research Approach

To resolve this problem, this research takes the following research approach:

Develop a self-reconfiguration mechanism

Validate this approach through case studies.

3.4. Summary

This chapter has described the problem statement and research approach used in this research for the self-reconfiguration.

CHAPTER VI

RECONFIGURATION MECHANISM

4.1. Self-Healing Systems

4.1.1. Software Architecture for Self-Healing Systems

The software architecture for self-healing systems [8] can be designed by means of self-healing components and self-healing connectors between the components. A self-healing component [9,10] is a component that is able to autonomously detect, isolate, and repair anomalies. A self-healing connector between the self-healing components also detects, isolates, and repairs anomalies. The architecture of a self-healing component enables a component to self-heal the anomalies that the component may come across at run-time, while the architecture of a self-healing connector makes it possible to self-heal the anomalies that may occur in message communication between components in a system.

Each self-healing component [11] can be structured into a layered architecture with the service layer and the healing layer. The service layer of a self-healing component provides services requested from neighboring components via connectors. The healing layer of a self-healing component takes care of anomalies detected in the service layer, notifying the status of self-healing to the healing layers of neighboring components. Similarly, each self-healing connector between self-healing components can be organized with two layers – communication layer and healing layer. The

connectors between self-healing components are used in message communication between service layers of components as well as between healing layers.

Figure 4.1 depicts part of the software architecture for a self-healing system, which is composed of self-healing components and self-healing connectors between the components. The Component1 and Component2 are self-healing components, which are structured into the service and healing layers. The ConnectorA and ConnectorB are self-healing connectors with the communication and healing layers.

Figure 4.1: Overview of Software Architecture for Self-Healing Systems

4.1.2. Self-healing Component Architecture

In the self-healing component architecture, the service layer is separated from the healing layer. Figure 4.2 depicts the self-healing component architecture. The service layer notifies the status of objects in the layer to the healing layer, whereas the healing layer detects, reconfigures, and repairs anomalies in the service layer using the

notification messages from the service layer.

The service layer of each self-healing component is composed of objects such as tasks (active or concurrent objects), passive objects accessed by tasks, and connectors between tasks. A task (active object or concurrent object), depicted using a thick outline for the object box in Figure 4.2, has its own thread of control, initiating actions that affect other tasks and passive objects [12]. Unlike a task, a passive object (e.g., an entity object) has no thread of control; thus it cannot initiate any task. A passive object is invoked by tasks and can invoke other passive objects. Since a passive object does not have its own thread, it performs its operations using the thread of the task that invoked the object. Tasks in a component can communicate with each other through connectors. On behalf of tasks that have threads, a connector between tasks synchronizes the message communication between the tasks. Passive objects and connectors between tasks, which do not have their threads, are depicted using thin outlines for the object boxes in Figure 4.2.

The healing layer of each self-healing component (Figure 4.2) is structured into four objects, such as Component Monitor, Reconfiguration Manager, Repair Manager, and Self-Healing Controller, which are responsible for detection, reconfiguration, and repair of anomalous objects in the service layer.

- **Component Monitor.** With messages notified by the service layer, the Component Monitor detects the anomalies of tasks, connectors between tasks, and passive objects accessed by tasks in the service layer using statecharts specifying the behavior of task threads.

Figure 4.2: Self-Healing Component Architecture

- **Reconfiguration Manager.** The Reconfiguration Manager generates reconfiguration plans for the service layer. It also carries out the plans to isolate anomalous objects from healthy objects in the service layer, and restores the repaired objects to the service layer.
- **Repair Manager.** The Repair Manager maintains information for repairing tasks, connectors between tasks, and passive objects accessed by tasks in the service layer. It generates repair plans in response to anomalous objects in the service layer, performing the plans to repair the objects.
- **Self-Healing Controller.** The Self-Healing Controller coordinates Component Manager, Reconfiguration Manager, and Repair Manager to conduct the self-healing mechanism including detection, reconfiguration, and repair of anomalous

objects. A self-healing controller of a self-healing component cooperates with self-healing controllers of other self-healing components for reconfiguration.

In the normal phase, the service layer of a self-healing component provides full functionality to service requests from other components and notifies the status of messages passing between objects in that layer to the healing layer. The notification messages are used to detect anomalous objects in the service layer. Once the healing layer detects an anomalous object in the service layer and presumes that the object needs to be repaired, it launches the self-reconfiguration mechanism.

4.1.3. Self-Healing Connector Architecture

A self-healing connector can be designed as a layered architecture with two layers – communication layer and healing layer. Figure 4.3 depicts the structure of a self-healing connector, which supports asynchronous message communication between components. Active objects are represented using a thick outline for the object box, while passive objects are depicted using a thin outline. The communication layer of a self-healing connector sends messages to or receives them from communication layers of other self-healing connectors, whereas the healing layer detects, reconfigures, and repairs anomalies in the communication layer. The communication layer of a self-healing connector consists of:

- **Call Stub.** The Call Stub packs a message and sends the packed message to a receiver connector via a network.

- **Return Stub.** The Return Stub unpacks a message delivered through the network and sends it to the destination component.
- **Incoming Message Queue.** The Incoming Message Queue stores messages delivered through network to a queue.
- **Outgoing Message Queue.** The Outgoing Message Queue stores messages, which should be sent to receiver components, to a queue.

The healing layer of a self-healing connector has:

- **Healing Manager.** The Healing Manager monitors objects in the communication layer of the connector using notification messages from the objects in the communication layer. This contains statecharts for each task thread – Call Stub task and Return Stub task - in the communication layer. The Healing Manager reconfigures objects in the communication layer against anomalies of objects, and repairs anomalous objects.

Figure 4.3: Self-Healing Connector Architecture

4.2. Anomaly Detection in Self-Healing Systems

An anomaly in the software architecture of a system can occur or be hidden in components and connectors between the components. A system is anomalous if components or connectors between the components in the architecture do not behave as specified. The specifications of components and connectors between the components are described using statecharts [13,14], which model the dynamic aspects of components and connectors between the components by means of states and transitions with events.

Anomalies in the software architecture with connectors are detected at two levels – at the level of each component and at the level of each connector between components [15]. Anomalies of tasks, passive objects accessed by tasks, and connectors between tasks in the service layer of a component are detected by the Component Monitor in a component (Figure 4.2), which traces the behavior of task threads in the component. To support detection of anomalous objects, a connector between tasks not only encapsulates a mechanism for synchronizing message communication between tasks, but also notifies messages passing between tasks to the healing layer.

Each task thread is monitored using notification messages from connectors between tasks and passive objects accessed by tasks. Whenever a connector between tasks and passive objects accessed by tasks are invoked by a task, they notify the component monitor in the healing layer of the component. Connectors between tasks and passive objects accessed by tasks also notify the component monitor when they finish their operations invoked by a task. The notification messages are used by the

component monitor to detect anomalies of tasks, connectors between tasks, and passive objects accessed by tasks.

To trace task threads in the service layer of a component, a component monitor encapsulates statecharts describing the behavior of each task thread. A task, connector between tasks, or a passive object accessed by tasks is anomalous if a statechart for the task thread does not execute as expected within a specified time interval.

The approach to anomaly detection in a self-healing system assumes that the network between connectors is reliable enough to detect/repair anomalies in delivering a message from a sender connector to a receiver connector. Otherwise, a network monitor may be added to a network between connectors to look after message delivery between sender and receiver connectors.

4.3. Approach to Dynamic Reconfiguration in Self-Healing Systems

A self-healing system is configured at the two levels – component level and connector level between components. The reconfiguration of anomalous objects in a component is performed by Reconfiguration Manager in the healing layer of the component (Figure 4.2). The Reconfiguration Manager in the healing layer of a component generates a reconfiguration plan against anomalous objects on the basis of the current configuration of the component. Using the plan, the Reconfiguration Manager blocks objects associated with the anomalous objects and, if needed, notifies the anomalies of objects to neighboring components affected from the reconfiguration of the paralyzed component. In response to this notification, the Reconfiguration Managers of the neighboring components also generate their reconfiguration plans and

undertake the plans to reduce impact from the paralyzed component.

When an anomalous object is detected in a connector between components, the Healing Manager in the healing layer of a connector (Figure 4.3) reconfigures objects in the communication layer. The Healing Manager blocks the anomalous object in the communication layer from other objects, notifying the anomaly to components associated with the anomalous object. The healing layers of the components notified generate reconfiguration plans that block message communication with other components via the connector containing the anomalous object. As the anomalous connector is recovered, the related components reconfigure their objects to communicate with other components through the repaired connector.

The approach to self-reconfiguration in this thesis does not include the atomic property of a task service. Before a task becomes anomalous, the task may have updated passive objects with its partial result during the process of a message. As the task is getting anomalous, the update to passive objects should be rolled back for system's consistency. But this is outside this thesis.

4.4. Reconfiguration of Objects in a Component

Reconfiguration of objects in the service layer of a self-healing component is performed differently on the basis of object types such as task, passive object accessed by tasks, and connector.

4.4.1. Anomalous Tasks

An anomalous task in the service layer of a self-healing component is reconfigured by blocking the incoming connectors to the task and outgoing connectors from it. The Reconfiguration Manager in the healing layer sends the incoming connectors a message requesting blocking the anomalous task not to read a message in the queues or buffers in the incoming connectors, while it sends the outgoing connectors a message requesting blocking the anomalous task not to add messages to the queues or buffers in the outgoing connectors. When receiving the messages from Reconfiguration Manager, the connectors respectively freeze message communication with the anomalous task so that the task is isolated from other objects.

Reconfiguration of an anomalous task is depicted in Figure 4.4, in which the service layer consists of three tasks (Task1, Task2, and Task3), two connectors between tasks (Connector1 and Connector2), and one passive object (Passive Object1) accessed by Task2 and Task3. The message sequence K1 through K8 in the service layer depicts message communication among objects in the service layer to provide services, while the message sequence S1 through S6 shows reconfiguration of objects in the service layer to isolate anomalous objects.

Suppose that Task1 in the service layer of Component1 is anomalous. The Component1 Monitor notifies the anomaly of Task1 to Component1 Self-Healing Controller (S1 in Figure 4.4), which requests reconfiguration of objects in the service layer of Component1 from Component1 Reconfiguration Manager (S2 in Figure 4.4). The Component1 Reconfiguration Manager generates a plan for isolating the anomalous Task1, sending messages (S3 and S3.1 in Figure 4.4) to the incoming

Connector1 to Task1 and the outgoing Connector2 from Task1. In response to the messages, the Connector1 and Connector2 allow Task1 not to access the connectors in order to isolate Task1 from other objects in the service layer of Component1.

The connectors between tasks in the service layer of a self-healing component support reconfiguration of anomalous tasks at runtime. A connector between tasks synchronizes message communication between two tasks. The functionality of a connector between tasks can be extended to support the reconfiguration of anomalous tasks in a self-healing component. More specifically, a connector between tasks can be classified to a message queue, a message buffer, and a message buffer and response connectors [12].

Figure 4.5 depicts a message queue self-healing connector, which provides the reconfiguration mechanism at runtime as well as the synchronization mechanism between tasks. The operations, send and receive, encapsulate the mechanism for synchronizing message communication between tasks, notifying Component Monitor (Figure 4.2) in the healing layer when both a sender task accesses the message queue (messageQueue with size, maxCount, in Figure 4.5) to send a message to a receiver task, and a receiver task reads a message from the queue (messageCount represents the current size of the queue). Those notification messages are used by Component Monitor to monitor the threads of sender and receiver tasks. The operations, blockSend() and blockReceive(), set sendStatus and receiveStatus variables (Figure 4.5) to Blocked respectively, which means a sender task cannot store a message in the queue and a receiver task cannot read a message from the queue. In Figure 4.4, Connector1 sets receiveStatus to Blocked and Connector2 sets SendStatus to Blocked

in response to the messages from Component1 Reconfiguration Manager if the connectors are message queue connectors supporting reconfiguration of tasks.

Figure 4.4: Dynamic self-configuration in Component1

Figure 4.5: Message Queue Self-Healing Connector

4.4.2. Anomalous Passive Object Accessed by Tasks

An anomalous passive object accessed by tasks is reconfigured by preventing the tasks from invoking operations of the passive object. Anomalies in a passive object accessed by tasks are detected when a task invokes an operation of the passive object. After the passive object notifies the invocation of an operation to Component Monitor (Figure 4.2), the operation may not be performed successfully so that the passive object cannot send the Component Monitor the next notification message showing the complete finish of the operation. The Component Monitor presumes that the passive object, more specifically, the operation invoked by a task, is anomalous. However, the repair of the anomalous passive object should be delayed until the tasks accessing the object are reconfigured by Reconfiguration Manager (Figure 4.2).

For isolating the anomalous passive object, the task invoked the anomalous operation of the passive object needs to be blocked and the thread of the task should be interrupted to terminate immediately. The Reconfiguration Manager creates a new thread for the task, but the thread cannot access the anomalous object because the thread checks the status of the passive object before invoking an operation provided by the passive object. The task blocked cannot call the operation again until the anomalous operation or the entire passive object is healed.

The operations performed by the terminated thread of the task may need to be rolled back to a previous status, just before the thread reads a new message from incoming connectors. It is to maintain consistency of the system. However, the rollback mechanism for consistency is outside the scope of this thesis.

Figure 4.6 shows the attributes and operations of Task2. The message “S3A

[Anomalous Passive Object1]: Block Task” is depicted in Figure 4.4 (at the middle of right) to block Task2 when Task2 invokes an anomalous operation to write a message in Passive Object1. The message blocks Task2 by setting a PassiveObject variable, representing the status of a passive object, to Blocked using blockPassiveObject() operation (Figure 4.6). Figure 4.7 depicts the specification of Task2. The Task2 cannot invoke “writeX()” operation of Passive Object1 if the status of passive object is “Blocked”.

<<task>> Task2
//Attributes for this task //Attribute for status of connectors - connectorStatus : {Blocked, Unblocked} := Unblocked; //Attribute for status of passive object accessed by this tasks - passiveObject : {Blocked, Unblocked} :=Unblocked; //Attribute for status of outgoing message queue in connector between components - outgoingMessageQueue: {Blocked, Unblocked} :=Unblocked;
//Operations for this task - initialize(); - writeX(); - sendX(); //Operations for configuration of connectors accessed by this task +blockConnector (); +unblockConnector(); //Operations for configuration of passive object accessed by this task +blockPassiveObject(); +unblockPassiveObject(); //Operations for configuration of connectors (between components) accessed by this task +blockOutgoingMessageQueue (); +unblockOutgoingMessageQueue();

Figure 4.6: Attributes and Operations of Task2

```

loop
  if (connectorStatus = Blocked)
  then
    wait;
  else
    aConnector.receive (message);
    Extract message name and any message parameters
      from message;
    if (passiveObjectStatus = Blocked)
    then
      wait;
    else
      writeX (parameters);
    if (outgoingMessageConnector = Blocked)
    then
      wait;
    else
      sendX (parameters);
  endloop;

```

Figure 4.7: Specification of Task2

The other tasks accessing the passive object with an anomalous operation, except for the task invoked an anomalous operation of a passive object, also need to be blocked in order to control the access to the object (e.g., S3A.1 in Figure 4.4). After an anomaly is found in the passive object, these tasks may not be blocked immediately to realize application properties such as safety. Suppose that a passive object accessed by two tasks - writer task and reader task - stores a list of floors at which an elevator should stop to drop off passengers in the elevator. The writer task stores the floor information to the passive object, while the reader task reads the information to go the destination floors in the list. When the “store” operation in the passive object is anomalous, the writer task should be blocked immediately, but the blocking of the reader task needs to be delayed enough for the elevator to drop off all passengers at the destination floors safely.

4.4.3. Anomalous Connectors between Tasks

An anomalous connector between tasks is reconfigured by blocking the sender task and receiver task accessing the connector. The sender task is blocked immediately when it invokes the anomalous “Send” operation of a connector to send a message to the receiver task. The receiver task may not be blocked immediately depending on the type of connectors such as a message queue connector (queue size is n), message buffer connector (buffer size is 1), and message buffer and response connector (both message buffer and response buffer sizes are one respectively). In the case of a message queue connector, the receiver task should be blocked after processing all the messages in the queue of the connector. It is to prevent messages already stored in the queue from being lost. In the other cases, the receiver task needs to be blocked immediately because there is no message in buffers.

On the other hand, a sender task and a receiver task should be blocked immediately when a receiver task calls an anomalous operation in a connector (e.g., receive operation in Figure 4.5). It is because new messages delivered by a sender task cannot be read by a receiver task any more. Messages already stored in the queue of a message queue connector may be backed up in repairing the connector, not to lose the messages.

The messages S3B and S3B.1 in Figure 4.4 block Task1 and Task2 respectively to isolate the anomalous Connector2. The message S3B.1 calls the “blockConnector()” operation in Task2, which sets the ConnectorStatus variable to a value “Blocked”. The ConnectorStatus variable represents the status of Connector2 from which Task2 reads messages. The Task2 checks the ConnectorStatus variable

whenever it reads a message (Figure 4.7). If the ConnectorStatus variable has been set to Blocked, the Task2 waits until the variable comes to be “Unblocked”, that is, the anomalous Connector2 is healed. Similarly, Task1 is blocked by the message S3B.

4.4.4. Impact of Reconfiguration of Component

The Reconfiguration Manager in the healing layer of a self-healing component maintains information related to reconfiguration of objects such as tasks, passive objects accessed by tasks, and connectors between tasks in the service layer. The information includes dependency relationships between objects within a component and in other components, which are used to infer the objects affected from an object’s anomaly. For example, in the elevator system, anomaly of Arrival Sensor Interface task in an Elevator Control component needs to force the Elevator Controller task to stop the elevator motor immediately due to the safety reason. The anomaly of Arrival Sensor Interface task should be also notified to the Elevator Scheduler component, which may reschedule other elevators to send an elevator to passengers at floors. The message sequence S4 through S6 in Figure 4.4 depicts the notification of anomaly of Component1 to other components.

4.5. Validation

The reconfiguration mechanism described in this paper has been applied to the self-healing Elevator System with multiple elevators, which is structured with the Elevator Control, Scheduler, and Floor components. Components communicate with

each other through asynchronous message queue connectors.

The Elevator Manager task in the Elevator Control component, which updates a list of floors at which an elevator has to stop, is reconfigured by blocking several connectors when it is anomalous. The Elevator Manager receives the floors to which passengers within an elevator want to go from the Elevator Button Interface task, and at which the elevator should visit to pick passengers up from the Scheduler component, respectively. Using a connector between components, the Elevator Manager also notifies the Scheduler component the list of destination floors of passengers within an elevator. The information on these floors is used by the Scheduler component to schedule elevators. When the Elevator Manager becomes anomalous, the receiving connectors should be blocked to isolate the anomalous Elevator Manager task. In addition, the anomaly of Elevator Manager needs to be notified to the Scheduler component, which should reschedule other elevators against the anomalous Elevator Manager in the elevator.

4.6. Conclusions

This chapter has described the reconfiguration mechanism for self-healing systems.

CHAPTER V

VALIDATION - CASE STUDY

5.1. Anomaly Detection

Anomaly detection in this thesis is based on exploiting the statecharts encapsulated in the component monitor of healing layer of a component. The statecharts describe the states and transitions of objects in the service layer of a component. The component monitor in the healing layer keeps track of every state and transition associated with each task, passive objects initiated by tasks, and connectors between objects.

The Component Monitor needs to compare between predefined normal states and the status messages arrived from objects in a component. Connectors and passive objects generate status messages that contain the name of task. If the system works as normal, connectors and passive objects make normal status messages and pass them to the component monitor in timely manner. However If a task works abnormally, related connector or passive object does not generate each normal status message. It means that the component monitor does not receive a proper status message or receive disordered status messages when it is compared with the next expecting status in predefined statechart of the task.

Figure 5.1 depicts an example for anomaly detection in a component.

Figure 5.1 : An example of anomaly detection in a component

Figure 5.2: Statechart for task1 in figure 5. 1

Figure 5.1 describes a component that has one task, one passive object, and two connectors in the service layer. To illustrate anomaly detection, the Component1 also has a component monitor which includes a monitor thread for Task1 and a detection message queue. The monitor thread checks every state of the Task1 using the statechart for the Task1. If the monitor thread cannot receive a normal status messages from the connectors and passive object in the service layer, the monitor thread detects an anomaly.

Task1 in Figure 5.1 reads a message in Connector1-Queue1 which came from outside of the component. Task1 executes the part of the receive operation. Just before the access to the receive operation, the Connector1-Queue1 sends a S.1 Read Invocation message to the Connector1-QueueD, which is a detection message queue for the Component Monitor. After the Task1 takes a message from the Connector1-Queue1, it also sends the S.2 Taken message to the Connector1-QueueD. The Task1 updates the passive object with intermediate information. While the Task1 updates the passive object, the passive object can notify the S.3 Update invocation message before updating and S.4 Updated message after updating the passive object. Similarly, the Connector1-Queue2 sends the S.5 Send Invocation and S.6 Placed message to the monitor when the Task1 calls the send operation. To describe connectors between components, Figure 5.2 is extended to Figure 5.3 by adding connectors.

Figure 5.3: Task architecture of elevator system with connectors

In Figure 5.3, the Elevator1 component includes two asynchronous connectors, two synchronous connectors, and two network connectors. Above all two network connectors are needed to implement communication with the other distributed component, Scheduler Component, on a network. These connectors are based on one to one socket communication on network. However this thesis does not cover reconfiguration of connectors between components. Additionally the Connector 2 is a queue that takes floor request messages from both local elevator button interface and the scheduler component.

Figure 5.4: Extended task architecture of elevator system with connectors

Figure 5.4 is derived from Figure 5.3 and Figure 5.1. The Elevator1 component has four detection connectors to trace normal activities of four tasks in the self-healing layer. As seen in Figure 5.1, all connectors and passive objects report their activities, which are described in the statecharts in the OComponent monitor.

Figure 5.5: Detailed design of Component monitor

The Component monitor consists of four task monitors that are the same as the number of tasks in Elevator1 component. Building concurrent self-healing component, each task is needed for a thread monitor. Each component encapsulates a statechart that describes the thread of each task.

Figure 5.6 Statechart for Elevator manager task

If the Elevator button interface task sends a floor request message to Queue2, the Elevator manger task will try to access the floor request message at the same queue. At that time, the Elevator manager task will reach the receive operation in the Queue2. The front part of the receive operation sends a detection message to the DQueue2. The first status message is the S1: Read invocation message (Figure 5.6).

After that, the Elevator manager task will take a floor request message from the Queue2. The end part of the receive operation should remove a floor request message from the Queue2. Just after removing, Queue2 will send the second status message, S2: Taken message, to DQueue2. This situation makes the transition from “Idle” state to “Receive the elevator floor request” state in Figure 5.6.

After processing the floor request message in Elevator manager task, it will update the contents of the passive object, Local elevator status & plan. In this case Elevator manager will add a floor request message to Local elevator status & plan for scheduling which floor has to be stopped. When the Elevator manager task adds a message to the object, the passive object lets DQueue2 have S3: Update invocation message. After that, Elevator manager will decide the next activity depending on whether Elevator controller is idle or not. If the Elevator controller is working, Elevator manger adds the next floor request message to the Local elevator status & plan. Otherwise the Elevator manger places a notification message on Buffer1 because Elevator controller needs to be awoken for processing the floor request message on Local elevator status & plan. The former case of the thread sends S4: Updated message and S7: Send invocation to DQueue2 since the Elevator manager does not need to send a notification message to Elevator controller. However the latter case of the thread processing needs to execute one more extra process. Therefore DQueue2 needs to get S.4 Updated, S.5 notification to buffer, and S.6 placed notification to buffer messages. The other part of the anomaly detection is the same as the Elevator Task.

If Elevator manager monitor does not receive S2: Taken message from the Queue2, it means that Queue2 cannot pop out the floor request message from the storage. Therefore this case was assumed as failing to providing method from the connector.

5.2. Reconfiguration

5.2.1. Introduction

This chapter addresses the reconfiguration in self-healing elevator system. The reconfiguration is carried out after the detection that is able to indicate that task, passive object, or connector between the tasks is anomalous using the statecharts. The reconfiguration mechanism localizes and disconnects healthy objects from the anomalous objects at runtime. This chapter describes three different cases of the reconfiguration - anomaly in task, anomaly in passive object, and anomaly in connector between tasks.

An anomalous task needs to be disconnected from its connectors. A task invokes receive() and send() operations provided by connectors to communicate with other tasks. If an anomaly in a task occurs, the connectors around the task are disconnected from the task by the reconfiguration manager in self-healing layer.

The anomaly in a passive object usually happens when a task invokes some operations of the passive object. The reconfiguration manager in the self-healing layer disconnects the anomalous task from the passive object.

The anomaly in a connector between tasks is blocked by disconnecting the tasks accessing the connector. The reconfiguration manager makes the tasks recognize

the anomaly in connector and change the related variable for the status of the connector.

5.2.2. Reconfiguration before the repair

This chapter describes the reconfiguration phase before the repair phase. After repairing anomalous objects in the self-healing mechanism, another reconfiguration is needed to restart and reinitiate temporarily suspended part of objects at runtime.

5.2.2.1. Elevator component

This Chapter deals with validation of dynamic reconfiguration in elevator component. Figure 5.7 depicts dynamic self-reconfiguration in Elevator component.

5.2.2.1.1 Anomalous tasks

Every object in the service layer would be managed by the reconfiguration manager in the healing layer. The reconfiguration manager has the name of each instance (object), which is used to initiate the reconfiguration of objects – task, connectors, and passive objects accessed by tasks. Figure 5.8 is part of Figure 5.7 for the Elevator Manager task, which accesses a queue, a buffer, and a passive object.

Figure 5.7: Dynamic self-reconfiguration in Elevator component1

Suppose that the Elevator Manager task has been detected as an anomalous task in Figure 5.6. The component monitor in the healing layer creates the detection notification message, “S.1[failed]: Notify” and sends it to the self-healing controller, which initiates a reconfiguration message. The reconfiguration manager generates a plan to send messages S.3C for Queue2 and S.3C.1 for Buffer1 so that the elevator manager is isolated from other objects.

Figure 5.8: Dynamic reconfiguration for Elevator manager task

Figure 5.9: Attribute and Operation of Queue2

The reconfiguration manager blocks the receive operation in the Queue2 connector and blocks the send operation in the Buffer1 connector. Figure 5.9 describes the detail attributes and operations of the Queue2 connector. The reconfiguration manager blocks sending messages from the Queue2 connector to the anomalous elevator manager task. To do this, the reconfiguration manager modifies the send attribute in Queue2, sendStatus in Figure 5.9, as 'Blocked'. This change occurs by using 'blockSend()' in Figure 5.9. Similarly, the Buffer1 connector has also same attributes and operations for blocking sending and receiving messages to the elevator manager.

5.2.2.1.2. Anomalous passive object accessed by tasks

This chapter deals with a passive object, "Local elevator status & plan", accessed by the Elevator manager and Elevator controller tasks. Figure 5.10 is part of Figure 5.7 that depicts the reconfiguration of anomalous Local elevator status & plan by blocking the invocation of the Elevator manager and from Elevator Controller tasks.

The reconfiguration manager generates a plan in which it sends "S.3E [Anomalous Local Status & Plan] Block Elevator Manager" to the elevator manager. In Figure 5.12, the reconfiguration manager modifies the attributes for Local elevator status & plan, Localelevatorstatus&planStatus, from 'Unblocked' to 'Blocked' by using the 'blockLocalelevatorstatus&plan()' operation. After this modification, the elevator manger is not able to perform "Update(message)" step in Figure 5.11 until the reconfiguration manager changes the attribute for Local elevator Status & plan from 'Blocked' to 'Unblocked'.

Figure 5.10: Dynamic reconfiguration for Local elevator status & plan

```

Loop
  if (Queue2Status = Blocked)
  then
    wait;
  else
    Queue2.receive(message);

    process message

    if (Localelevatorstatus&planStatus = Blocked)
    then
      wait;
    else
      Update(message);
      if (Localelevatorstatus&plan.IdleCheck() = True)
      then
        if (Buffer1Status = Blocked)
        then
          wait;
        else
          Buffer1.send(parameters);
        else
          outGoingQueue1.send(parameters);
      end if
    end if
  end if
Endloop;

```

Figure 5.11: Specification of Elevator Manager

<<task>> Elevator manager
<pre> //Attributes for Elevator manager ... //Attributes for status of Queue2 Queue2Status : {Blocked, Unblocked} := Unblocked; //Attributes for status of Local elevator status & plan Local elevator status & planStatus : {Blocked, Unblocked} := Unblocked; //Attributes for status of Buffer1 Buffer1Status : {Blocked, Unblocked} := Unblocked; //Attributes for status of outGoingQueue1 outGoingQueue1 : {Blocked, Unblocked} := Unblocked; </pre>
<pre> //Operation for Elevator manager -initialize() -update() //Operation for configuration of Queue2 accessed by Elevator manager +blockQueue2() +unblockQueue2() //Operation for configuration of Buffer1 accessed by Elevator manager +blockBuffer1() +unblockBuffer1() //Operation for configuration of Local elevator status & plan accessed by Elevator manager +blockLocalelevatorstatus&plan() +unblockLocalelevatorstatus&plan() //Operation for configuration of outGoingQueue1 accessed by Elevator manager +blockoutGoingQueue1() +unblockoutGoingQueue1() </pre>

Figure 5.12: Attributes and Operation of Elevator manager

The elevator controller is also blocked by “S.3E.1 [Anomalous Local Status & Plan] Block Elevator Controller”. The similar reconfiguration process is applied to the elevator controller. However, blocking the Elevator Controller needs to be delayed to finish safely the execution of already stored floor requests in the Local Elevator status & Plan.

5.2.2.1.3. Anomalous connectors between tasks

This chapter describes the reconfiguration of the Queue2 connector between the Elevator manger task and the Elevator button interface task. In Figure 5.6, suppose that “S2 Taken message” is corrupted by some reason. S2 Taken message is about taking a message from the connector by the Elevator Manager.

The reconfiguration manager makes the Elevator Manager not receive messages from the Queue2 connector. This step is depicted as S.3B.1 [Anomalous Queue2] Block Elevator Manager in Figure 5.11. The reconfiguration manager changes the status of receiving attribute of Elevator Manager, Queue2Status attribute in Figure 5.12, from initially unblocked attribute to blocked attribute. This blocked attribute will affect to perform inside of condition, ‘if(Queue2 Status = Blocked)’, in Figure 5.12.

The other part of reconfiguration is blocking the sender task. The Reconfiguration Manager blocks sending messages from the Elevator button interface to the Queue2 connector. This step is illustrated by S.3B [Anomalous Queue2] Block Elevator Button Interface in Figure 5.13. The Reconfiguration Manager also modifies the send attribute in the Elevator Button Interface task.

Figure 5.13: Dynamic reconfiguration for Queue2 in Elevator component

5.2.2.2. Scheduler component

This chapter describes validation of dynamic self-reconfiguration in Scheduler component. Figure 5.14 illustrates self-reconfiguration in scheduler component.

5.2.2.2.1. Anomalous tasks

Suppose that Elevator status & plan server task in Scheduler component is in anomaly. The Elevator status & plan sever task is linked with two connectors in Figure 5.15. After reconfiguration, Elevator status & plan server task will be isolated from those two connectors.

Figure 5.14: Dynamic reconfiguration for Scheduler component

Figure 5.15 depicts the dynamic reconfiguration for Elevator status & plan server task. The reconfiguration manager generates a plan to send a message S.3D and S.3D.1 (Figure 5.15) for Queue3 and Queue4 so that the Elevator status & plan task is isolated from other objects. Queue3 and Queue4 have the same attributes as in Figure 5.9. Therefore attribute for blocking receive operation will be changed as “Blocked”. From this isolation, Elevator status & plan server task cannot perform update commitments and arrival messages from the each Elevator components in Overall status & plan object until repair step is done.

Figure 5.15: Dynamic reconfiguration for Elevator status & plan server task

5.2.2.2.2. Anomalous passive object accessed by tasks

We can apply the same approach to Overall Elevator status & plan passive object in Figure 5.16. Each task in Scheduler component also has blocking attributes for isolating the Overall Elevator status & plan passive object. After receiving the S.3C message in Figure 5.16, the attributes for Overall elevator status & plan passive object will be changed from “Unblocked” to “Blocked”. After that, tasks in the scheduler will not update and access Overall status & plan object.

Figure 5.16: Dynamic reconfiguration for Local elevator status & plan

5.2.2.2.3. Anomalous connectors between tasks

If the Queue3 in Scheduler component was detected as anomalous object, Reconfiguration manager only need to change the attribute for Queue3 from “Unblocked” to “Block”. Since Elevator status & plan server task is the only one task that needs to access Queue3 in Figure 5.17, Reconfiguration manager will plan to send a message S.3E (Figure 5.17) to the Elevator Status & Plan Server task that is related to Queue3. If Queue4 is still working well, Elevator Status & plan server task needs to perform its job for Elevator controllers under the blocking of the commitment message from Queue3.

Figure 5.17: Dynamic reconfiguration for Queue3 in Scheduler component

5.2.2.3. Notification among components.

After reconfiguration of Elevator Manager is done, the Reconfiguration Manager passes anomaly notification, S.4 Anomaly Notification, in the Elevator Component1 (Figure 5.7) to the Self-healing controller. Self-healing controller will also send notification message, S.5 Notification in Figure 5.7, 5.13), to the Self-healing controller in the Scheduler Component through the network resources. The Self-healing controller in Elevator Component1 needs to alert anomalous behavior of tasks inside the component to the Scheduler for changing floor request scheduling. If the Elevator manager task is reconfigured, the Elevator cannot service any floor request through the Scheduler component. Therefore an anomalous task in the Elevator component1 makes the Scheduler component change the floor scheduling for servicing floor request consistently. It is accomplished by allocating the floor request to Elevator component2 or 3 except Elevator component1 while reconfiguration plan is going on. S.6 [Anomalous Elevator Manager] Block Receiver and Block Sender in Figure 5.18 demonstrate blocking anomaly message communication between Elevator manager task and Scheduler tasks in Scheduler component through the network resource. S.6.1 anomaly notification for Elevator1 and Reschedule plan in Figure 5.18 is for excluding Elevator1 component when the scheduler allocates a request to elevators. After this step, scheduler will recalculate next closest elevator without Elevator1 component and the other Elevator will perform floor requests.

Figure 5.18:Dynamic reconfiguration for inter components

5.3. Repair

This section deals with repair after reconfiguration which is implemented by Java reflection mechanism at runtime. Java reflection mechanism provides a way to go through or modify the runtime activity of the application in Java Virtual Machine (JVM). The substitution of the original instance of a specific task class is the core of this chapter

The reconfiguration manager notifies the finish of the reconfiguration to the self-healing controller, which initiates the repair manger by informing the class name of the reconfigured task. The elevator manager is repaired. All Classes in application were loaded on JVM when those were compiled. The repair manager looks up the class name of the elevator manger task in JVM. By the name of the class, the repair

manager instantiates a substitution object of the current elevator manager. Newly instantiated object of elevator manager has the same attributes and operations except the name for distinguishing it from the former one. In other words, it is totally a new instance, however it does not contain any information of previous instance of elevator manager task.

5.4. Reconfiguration after repair

This section illustrates reconfiguration phase after repair phase. In this phase the temporarily suspended object restarts its own work after reconfiguration. Most of the reconfiguration part is the reverse process of the previous reconfiguration before repair.

Figure 5.19: dynamic reconfiguration after repair for Elevator1 component

The repair manager needs to inform the finishing of the objects to the self-healing controller. The self-healing controller sends a notification, S.1 Reconfiguration for restart in Figure 5.19, to the reconfiguration manager. All process of the reconfiguration after repair followed all step of S.2 in Figure 5.19. As the reconfiguration before repair, the reconfiguration manager performs unblocking operation for restart. In Figure 5.9, 5. 12) related attributes for objects which were blocked temporarily in reconfiguration before repair phase would be changed to original status, 'Unblocked' in Figure 5.9, 5. 12).

After changing the attributes, reconfiguration will wake the each suspended task to work again. The related tasks will pass the checking condition part of those attributes in the specifications. The elevator manager will check the status of the Queue2 as 'unblocked'; the elevator manager will perform the next line in Figure 5.11.

All related object in the elevator component works again, and the self-healing controller in Elevator will let the self-healing controller in Scheduler component know the finishing of the reconfiguration in Elevator component for reconfiguration in Scheduler component. As seen in chapter 5.2.2.3, there are some suspended objects of the Scheduler component in the service layer. The reconfiguration manager in Scheduler component needs to make those objects resume.

5.5 Conclusions

This chapter has described the case study for the elevator system as a validation of this research.

CHAPTER VI

CONCLUSION

This paper has described an approach to self-reconfiguration of a self-healing system in which anomalies may be detected in components and connectors between the components. The self-reconfiguration approach in this paper is part of a self-healing mechanism including detection, reconfiguration, and repair of anomalies in a system. The self-reconfiguration scenarios are classified on the basis of object types, such as task, passive object assessed by tasks, and connector between tasks.

The approach to self-reconfiguration in this paper makes anomalous objects easy to be localized, relatively compared to other approaches [4, 5]. It attributes to the component architecture, which is structured in the center of connectors between tasks. The connectors provide a way of blocking anomalous tasks immediately, preventing the impact of the anomalies from being spread to other objects within the component and other components.

The self-reconfiguration mechanism in this paper can be extended to including atomic property of anomalous tasks. The basic idea about this is that a transaction mechanism can be added to passive objects assessed by tasks and connectors between tasks in a self-healing component.

REFERENCES

- [1] Eric M. Dashofy, Andre van der Hoek, and Richard N. Taylor, "Towards Architecture-based Self-Healing Systems," Workshop on Self-healing systems, Proceedings of the first workshop on Self-healing systems, Charleston, SC, November 18-19, 2002.
- [2] David Garlan, Shang-Wen Cheng, and Bradley Schmerl, "Increasing System Dependability through Architecture-based Self-repair," in *Architecting Dependable Systems*. R. de Lemos, C. Gacek, A. Romanovsky (Eds), Springer-Verlag, 2003.
- [3] IBM, "An architectural blueprint for autonomic computing," IBM and autonomic computing, April 2003.
- [4] Jeff Kramer and Jeff Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE TSE* 16, 11, November 1990.
- [5] H. Gomaa and M. Hussein, "Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures", *Proc. Fourth Working IEEE/IFIP Conference on Software Architecture*, Oslo, Norway, June, 2004.
- [6] João Paulo A. Almeida, Maarten Wegdam, Luís Ferreira Pires, and Marten van Sinceren, "An approach to dynamic reconfiguration of distributed systems based on object-middleware," *Proceedings of 19 Brazilian Symposium on Computer Networks*, 2001.
- [7] Halima, R. B., Drira, K., and Jmaiel, M. 2007. A QoS-Driven Reconfiguration Management System Extending Web Services with Self-Healing Properties. In *Proceedings of the 16th IEEE international Workshops on Enabling Technologies: infrastructure For Collaborative Enterprises* (June 18 - 20, 2007).
- [8] Michael E. Shin and Daniel Cooke, "Connector-Based Self-Healing Mechanism for Components of a Reliable System," Workshop on Design and Evolution of Autonomic Application Software (A Workshop of ICSE2005), St. Louis, Missouri, USA, May 21, 2005.
- [9] Paulo Asterio de C. Guerra and Rogerio de Lemos, "An Idealized Fault-Tolerant Architectural Component," Workshop on Architecting Dependable Systems of ICSE'02, Orlando, FL, May 25, 2002.
- [10] Philip Koopman, "Elements of the Self-Healing System Problem Space," Workshop on Software Architectures for Dependable Systems (WADS2003), ICSE'03 International Conference on Software Engineering, Portland, Oregon, May 3-11, 2003

- [11] Michael E. Shin, “Self-Healing Component in Robust Software Architecture for Concurrent and Distributed Systems,” *Science of Computer Programming*, Vol. 57, No. 1, 2005, pp 27-44.
- [12] Hassan Gomaa, “Designing Concurrent, Distributed, and Real-Time Applications with UML,” Addison-Wesley, 2000.
- [13] G. Booch, J. Rumbaugh, I. Jacobson, “The Unified Modeling Language User Guide”, Addison Wesley, Reading MA, 1999.
- [14] J. Rumbaugh, G. Booch, I. Jacobson, “The Unified Modeling Language Reference Manual,” Addison Wesley, Reading MA, 1999.
- [15] Michael E. Shin and Yan Xu, “Detection of Anomalies in a Software Architecture with Connectors,” *International Workshop on System/Software Architectures (WSSA05)*, Las Vegas, Nevada, USA, June 27-30, 2005.

PERMISSION TO COPY

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Texas Tech University or Texas Tech University Health Sciences Center, I agree that the Library and my major department shall make it freely available for research purposes. Permission to copy this thesis for scholarly purposes may be granted by the Director of the Library or my major professor. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my further written permission and that any user may be liable for copyright infringement.

Agree (Permission is granted.)

Jung Hoon An

10-17-2008

Student Signature

Date

Disagree (Permission is not granted.)

Student Signature

Date