SELF-MANAGEMENT OF COTS COMPONENTS-BASED
SYSTEMS USING WRAPPERS

by

Fernando G. Paniagua, B.S.

A Dissertation

In

COMPUTER SCIENCE

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

Doctor of Philosophy in Computer Science

Approved

Eonsuk (Michael) Shin

Daniel Cooke

Richard Watson

Yu Zhuang

Fred Hartmeister
Dean of the Graduate School

May, 2009

Copyright 2009, Fernando G. Paniagua

ACKNOWLEDGMENTS

I would like to express my appreciation to my dissertation advisor, Dr. Michael (Eonsuk) Shin. His guidance helped me learn not only the main topics of software engineering, but also led me to develop as a professional. I thank him for his support, mentoring and patience while I was writing my dissertation. I also want to express my gratitude to Dr. Per Andersen for his valuable help in the implementation of the first case study. I would like to thank all the members of my dissertation committee, Dr. Daniel Cooke, Dr. Richard Watson, and Dr. Yu Zhuang for their invaluable feedback.

I would like to thank Dr. Daniel Cooke for giving me the opportunity to start my teaching career as a Teaching Assistant and then as a Lecturer in the Computer Science Department.

I am also grateful to the friends I made during my career at Texas Tech University. It was also their support and encouragement that kept me on track all these years. I would like to thank Verónica, Roberto, Ximena, Jaime, Leticia, Edgar, René, and Hilda, for their friendship, their companionship, and their laughter.

I want to express my gratitude to my lovely wife Silvia and my daughter Flor. Silvia, you have always been and will always be the love of my life. Thank you for your understanding, patience, and support in the good and in the bad times. Thank you Flor for your love, and for making my life happier with your laugh and your sense of humor. I would like to also thank my brother Oscar, his wife Kristina and their children, Gabriel, Grace and Annie for their love and support during my graduate studies. My parents also deserve recognition for encouraging me to pursue my goals, and for giving me the best gift in life, a good education.

TABLE OF CONTENTS

	Page
ACKNOWLEDGMENTS	ii
ABSTRACT.....	ix
LIST OF TABLES	x
LIST OF FIGURES	xvii
CHAPTER 1: INTRODUCTION	1
1.1 Background	1
1.2 Problem Statement and Research Approach.....	2
1.3 Importance and Rationale of This Research	3
1.4 Terminology.....	4
1.5 Organization.....	4
CHAPTER 2: RELATED RESEARCH	6
2.1 Introduction.....	6
2.2 IBM Autonomic Computing	6
2.3 Software Fault Tolerance Approaches.....	7
2.3.1 Single-Version Software Fault Tolerance Techniques	7
2.3.2 Multiple-Version Software Fault Tolerance Techniques.....	9
2.4 Mediators and Obligations.....	13
2.5 Idealized C2 Components (iC2C).....	14
2.6 Protective Wrappers.....	15

2.7 Self-Healing Approaches	17
2.7.1 Elements of the Problem Space	17
2.7.2 Architecture-Based Approaches	19
2.8 Comparison Among the Approaches	22
CHAPTER 3: PROBLEM STATEMENT AND RESEARCH APPROACH.....	24
3.1 Introduction.....	24
3.2 Problem Statement.....	24
3.3 Research Approach	25
3.4 Summary	26
CHAPTER 4: DETECTION MECHANISM TO DETECT COTS COMPONENTS ANOMALIES	27
4.1 Introduction.....	27
4.2 Software Architecture for Self-Managed COTS Component-Based Systems	27
4.3 Detection Mechanism	29
4.4 High-Level View of Algorithm for Detection Mechanism.....	35
4.5 Summary	37
CHAPTER 5: RECONFIGURATION MECHANISM TO MINIMIZE THE IMPACT FROM THE ANOMALIES DETECTED	38
5.1 Introduction.....	38
5.2 Reconfiguration Mechanism.....	39
5.2.1 Reconfiguration of Critical Functions	41
5.2.2 Reconfiguration of Non-Critical Functions	50
5.3 High-Level View of Algorithm for Reconfiguration Mechanism	53
5.4 Summary	55

CHAPTER 6: REPAIR MECHANISM.....	56
6.1 Introduction.....	56
6.2 Repair Techniques	57
6.3 Repair Mechanism	58
6.4 High-Level View of Algorithm for Repair Mechanism	60
6.5 Summary	61
CHAPTER 7: DESIGN OF A COTS COMPONENT WRAPPER FOR SELF-MANAGING COTS COMPONENTS	62
7.1 Introduction.....	62
7.2 Requirement for Designing a COTS Component Wrapper	63
7.2.1 COTS Component Wrapper Functionalities to Support the Detection Mechanism	64
7.2.2 COTS Component Wrapper Functionalities to Support the Reconfiguration Mechanism	65
7.2.3 COTS Component Wrapper Functionalities to Support the Repair Mechanism	66
7.3 Design of COTS Component Wrappers	67
7.3.1 COTS Modified Interface	67
7.3.2 COTS Monitor	72
7.3.3 Reconfiguration Manager	74
7.3.4 Repair Manager.....	77
7.3.5 Wrapper Controller	78
7.4 Interaction of the Objects in the COTS Component Wrapper	79
7.4.1 Detection Mechanism	79
7.4.2 Reconfiguration Mechanism.....	85

7.4.3 Repair Mechanism	91
7.5 Summary	93
CHAPTER 8: DISTRIBUTED ELEVATOR SYSTEM: CASE STUDY	95
8.1 Introduction.....	95
8.2 Problem Description	96
8.3 Software Architecture	97
8.4 Implementation of the Distributed Elevator System.....	99
8.4.1 Floor COTS Component Wrapper	100
8.4.2 Scheduler COTS Component Wrapper.....	104
8.4.3 Elevator COTS Components Wrapper.....	109
8.5 Validation of the Distributed Elevator System	116
8.5.1 Hardware and Software Specifications	116
8.5.2 Validation of the Detection Mechanism's Algorithms	116
8.5.3 Validation of the Reconfiguration Mechanism's Algorithms	122
8.5.4 Validation of the Repair Mechanism's Algorithms	134
8.6 Summary	137
CHAPTER 9: DISTRIBUTED FACTORY AUTOMATION SYSTEM: A CASE STUDY	139
9.1 Introduction.....	139
9.2 Problem Description	140
9.3 Software Architecture	141
9.4 Implementation of the Distributed Factory Automation System.....	145
9.4.1 Production Manager COTS Component Wrapper	146

9.4.2	Process Engineer COTS Component Wrapper	152
9.4.3	Receiving Workstation Controller COTS Component Wrapper.....	156
9.4.4	Line Workstation Controller COTS Component Wrapper.....	160
9.4.5	Shipping Workstation Controller COTS Component Wrapper.....	166
9.5	Validation of the Distributed Factory Automation System	170
9.5.1	Hardware and Software Specifications.....	170
9.5.2	Validation of the Detection Mechanism's Algorithms	171
9.5.3	Validation of the Reconfiguration Mechanism's Algorithms	178
9.5.4	Validation of the Repair Mechanism's Algorithms	191
9.6	Summary	194
CHAPTER 10: CONTRIBUTIONS AND FUTURE RESEARCH.....		196
10.1	Introduction	196
10.2	Research Contributions	197
10.3	Future Research	199
10.3.1	Enhancements to the COTS Monitor.....	199
10.3.2	Enhancements for Hard Real-Time Systems	200
10.3.3	Enhancements to Decrease System Complexity.....	200
10.3.4	Extended Validation.....	201
10.4	Summary.....	201
REFERENCES		203

APPENDIX A: IMPLEMENTATION OF A COTS COMPONENT WRAPPER	205
A.1 Introduction.....	205
A.2 Implementation of the Detection Mechanism.....	206
A.3 Implementation of the Reconfiguration Mechanism	209
A.4 Implementation of the Repair Mechanism.....	217
APPENDIX B: TESTING OF THE DISTRIBUTED ELEVATOR SYSTEM ..	220
B.1 Introduction.....	220
B.2 Testing of the Detection Mechanism	220
B.3 Testing Result of the Detection Mechanism.....	224
B.4 Testing of the Reconfiguration Mechanism.....	227
B.5 Testing Result of the Reconfiguration Mechanism	242
B.6 Testing of the Repair Mechanism	243
B.7 Testing Result of the Repair Mechanism.....	244
APPENDIX C: TESTING OF THE DISTRIBUTED FACTORY AUTOMATION SYSTEM.....	246
C.1 Introduction.....	246
C.2 Testing of the Detection Mechanism	246
C.3 Testing Result of the Detection Mechanism.....	250
C.4 Testing of the Reconfiguration Mechanism.....	255
C.5 Testing Result of the Reconfiguration Mechanism	274
C.6 Testing of the Repair Mechanism	277
C.7 Testing Result of the Repair Mechanism.....	279

ABSTRACT

Based on a survey of self-healing and self-management methods, current approaches address self-management of COTS component-based systems in an ad-hoc way. There is no self-management mechanism that addresses self-management of COTS component-based systems in a systematic way. Hence, self-managed COTS components need to be capable of detecting anomalies in COTS components, reconfiguring COTS components against the anomalies, and repairing the anomalies in COTS components.

This research develops an approach to designing COTS component-based software systems in which each COTS component is self-managed by a wrapper to improve dependability of systems. The wrapper for self-managing a COTS component detects the anomalies of the component; it reconfigures the anomalous component at run-time; and it repairs the anomalous component. In this research, the wrapper architecture is also described to support self-management of COTS components in terms of static and dynamic aspects of software architecture. The wrappers co-operate with each other to reconfigure COTS components, so that they make the system more reliable.

The approach presented in this research is applied to two case studies, a distributed elevator system and a distributed factory automation system in order to validate not only the detection, reconfiguration and repair mechanisms but also the software architecture of the wrapper.

LIST OF TABLES

	Page
Table 2.1: Comparison Among the Approaches Surveyed	23
Table 5.1: Scheduler Reconfiguration Table	44
Table 5.2: Reconfiguration Plan for Service Request Function in Scheduler Component	45
Table 5.3: Floor Reconfiguration Table	46
Table 5.4: Reconfiguration Plan for Service Request Function in Floor Component.....	47
Table 5.5: Elevator Reconfiguration Table	47
Table 5.6: Reconfiguration Plan for Update Status Plan Function in Elevator Component.....	48
Table 5.7: Reconfiguration Plan for Update Elevator Status Function in Elevator Component.....	49
Table 5.8: Reconfiguration Plan for Off Floor Lamp Function in Floor Component.....	52
Table 5.9: Reconfiguration Plan for Off Floor Lamp Function in Elevator Component.....	53
Table 6.1: Floor Repair Techniques	57
Table 8.1: Floor Operation Status Table	100
Table 8.2: Floor Operation Request Parameter Table	100
Table 8.3: Floor Operation Return Type Table	101
Table 8.4: Floor Successor Operation Table	101

Table 8.5: Floor Function Status Table	101
Table 8.6: Floor Operation Statechart Table	102
Table 8.7: Floor Operation Statechart Information Table	102
Table 8.8: Floor Operation Criticality Table	103
Table 8.9: Floor Operation Reconfiguration Status Table	103
Table 8.10: Floor Reconfiguration Table.....	104
Table 8.11: Floor Repair Techniques Table.....	104
Table 8.12: Scheduler Operation Status Table.....	105
Table 8.13: Scheduler Operation Request Parameter Table	105
Table 8.14: Scheduler Operation Return Type Table	106
Table 8.15: Scheduler Successor Operation Table	106
Table 8.16: Scheduler Function Status Table	106
Table 8.17: Scheduler Operation Statechart Table	107
Table 8.18: Scheduler Operation Statechart Information Table	107
Table 8.19: Scheduler Operation Criticality Table	108
Table 8.20: Scheduler Operation Reconfiguration Status Table.....	108
Table 8.21: Scheduler Reconfiguration Table	109
Table 8.22: Scheduler Repair Techniques Table	109
Table 8.23: Elevator Operation Status Table	110
Table 8.24: Elevator Operation Request Parameter Table.....	110
Table 8.25: Elevator Operation Return Type Table.....	111
Table 8.26: Elevator Successor Operation Table.....	111
Table 8.27: Elevator Function Status Table.....	112

Table 8.28: Elevator Operation Statechart Table	113
Table 8.29: Elevator Operation Statechart Information Table	114
Table 8.30: Elevator Operation Criticality Table	114
Table 8.31: Elevator Operation Reconfiguration Status Table	115
Table 8.32: Elevator Reconfiguration Table	115
Table 8.33: Elevator Repair Techniques Table	115
Table 8.34: Application-Independent Algorithms for Detection Mechanism ...	120
Table 8.35: Application-Independent Algorithms for Reconfiguration Mechanism	125
Table 8.36: Floor Reconfiguration Plan for Floor Button Request Function	127
Table 8.37: Elevator Reconfiguration Plan for Off Floor Lamp Function	128
Table 8.38: Elevator Reconfiguration Plan for On Off Direction Lamp Function	129
Table 8.39: Floor Reconfiguration Plan for Off Floor Lamp Function	131
Table 8.40: Floor Reconfiguration Plan for On Off Direction Lamp Function .	132
Table 8.41: Application-Independent Algorithms for Repair Mechanism	136
Table 9.1: Production Manager Operation Status Table	146
Table 9.2: Production Manager Operation Request Parameter Table	147
Table 9.3: Production Manager Operation Return Type Table	147
Table 9.4: Production Manager Successor Operation Table	147
Table 9.5: Production Manager Function Status Table	148
Table 9.6: Production Manager Operation Statechart Table	149
Table 9.7: Production Manager Operation Statechart Information Table	150
Table 9.8: Production Manager Operation Criticality Table	150

Table 9.9: Production Manager Operation Reconfiguration Status Table	151
Table 9.10: Production Manager Reconfiguration Table.....	151
Table 9.11: Production Manager Repair Techniques Table.....	152
Table 9.12: Process Engineer Operation Status Table	152
Table 9.13: Process Engineer Operation Request Parameter Table.....	153
Table 9.14: Process Engineer Operation Return Type Table.....	153
Table 9.15: Process Engineer Successor Operation Table.....	153
Table 9.16: Process Engineer Function Status Table.....	153
Table 9.17: Process Engineer Operation Statechart Table.....	154
Table 9.18: Process Engineer Operation Statechart Information Table.....	154
Table 9.19: Process Engineer Operation Criticality Table.....	154
Table 9.20: Process Engineer Operation Reconfiguration Status Table	155
Table 9.21: Process Engineer Reconfiguration Table.....	155
Table 9.22: Process Engineer Repair Techniques Table.....	155
Table 9.23: Receiving Workstation Controller Operation Status Table	156
Table 9.24: Receiving Workstation Controller Operation Request Parameter Table	157
Table 9.25: Receiving Workstation Controller Operation Return Type Table..	157
Table 9.26: Receiving Workstation Controller Successor Operation Table	157
Table 9.27: Receiving Workstation Controller Function Status Table.....	157
Table 9.28: Receiving Workstation Controller Operation Statechart Table	158
Table 9.29: Receiving Workstation Controller Operation Statechart Information Table	158
Table 9.30: Receiving Workstation Controller Operation Criticality Table.....	159

Table 9.31: Receiving Workstation Controller Operation Reconfiguration Status Table.....	159
Table 9.32: Receiving Workstation Controller Reconfiguration Table	160
Table 9.33: Receiving Workstation Controller Repair Techniques Table	160
Table 9.34: Line Workstation Controller Operation Status Table	161
Table 9.35: Line Workstation Controller Operation Request Parameter Table	161
Table 9.36: Line Workstation Controller Operation Return Type Table.....	162
Table 9.37: Line Workstation Controller Successor Operation Table	162
Table 9.38: Line Workstation Controller Function Status Table	162
Table 9.39: Line Workstation Controller Operation Statechart Table	163
Table 9.40: Line Workstation Controller Operation Statechart Information Table	164
Table 9.41: Line Workstation Controller Operation Criticality Table.....	164
Table 9.42: Line Workstation Controller Operation Reconfiguration Status Table	165
Table 9.43: Line Workstation Controller Reconfiguration Table	165
Table 9.44: Line Workstation Controller Repair Techniques Table.....	165
Table 9.45: Shipping Workstation Controller Operation Status Table	166
Table 9.46: Shipping Workstation Controller Operation Request Parameter Table	167
Table 9.47: Shipping Workstation Controller Operation Return Type Table....	167
Table 9.48: Shipping Workstation Controller Successor Operation Table.....	167
Table 9.49: Shipping Workstation Controller Function Status Table.....	167
Table 9.50: Shipping Workstation Controller Operation Statechart Table.....	168

Table 9.51: Shipping Workstation Controller Operation Statechart Information Table	168
Table 9.52: Shipping Workstation Controller Operation Criticality Table.....	168
Table 9.53: Shipping Workstation Controller Operation Reconfiguration Status Table.....	169
Table 9.54: Shipping Workstation Controller Reconfiguration Table.....	169
Table 9.55: Shipping Workstation Controller Repair Techniques Table.....	170
Table 9.56: Application-Independent Algorithms for Detection Mechanism ...	174
Table 9.57: Application-Independent Algorithms for Reconfiguration Mechanism.....	181
Table 9.58: Production Manager Reconfiguration Plan for Create Part Orders Function	182
Table 9.59: Receiving Workstation Controller Reconfiguration Plan for Part Request Function	184
Table 9.60: Line Workstation Controller Reconfiguration Plan for Next Operation Request Function	185
Table 9.61: Shipping Workstation Controller Reconfiguration Plan for Part Order Complete Function	186
Table 9.62: Application-Independent Algorithms for Repair Mechanism	193
Table B.1: Scheduler Reconfiguration Plan for Service Request Function	228
Table B.2: Floor Reconfiguration Plan for Service Request Function.....	230
Table B.3: Elevator Reconfiguration Plan for Update Status Plan Function	231
Table B.4: Elevator Reconfiguration Plan for Update Elevator Status Function	231
Table B.5: Elevator Reconfiguration Plan for Schedule Request Function	235
Table B.6: Floor Reconfiguration Plan for Schedule Request Function	236
Table C.1: Process Engineer Reconfiguration Plan for Operation Request Function	255

Table C.2: Production Manager Reconfiguration Plan for Operation Request Function	257
Table C.3: Receiving Workstation Controller Reconfiguration Plan for Start Part Function.....	258
Table C.4: Production Manager Reconfiguration Plan for Start Part Function	259
Table C.5: Line 1 Workstation Controller Reconfiguration Plan for Part Request Function	260
Table C.6: Line Workstation Controller Reconfiguration Plan for Part Coming Function	264
Table C.7: Predecessor Workstation Controller Reconfiguration Plan for Part Coming Function.....	265
Table C.8: Successor Workstation Controller Reconfiguration Plan for Part Request Function	266
Table C.9: Shipping Workstation Controller Reconfiguration Plan for Part Coming Function	271
Table C.10: Predecessor Workstation Controller Reconfiguration Plan for Part Coming Function	273

LIST OF FIGURES

	Page
Figure 4.1: Software Architecture with Self-Managed COTS Components	29
Figure 4.2: Floor and Scheduler COTS Component of the Elevator System	31
Figure 4.3: Statechart for Floor Button Request Function	33
Figure 4.4: Statechart for Service Request Function	34
Figure 5.1: Reconfiguration for a Critical Function of the Elevator System	42
Figure 5.2: Reconfiguration for a Non-Critical Function of the Elevator System	51
Figure 7.1: COTS Component Wrapper Architecture for Self-Managed COTS Components	67
Figure 7.2: Operation Request Parameter Entity Class	69
Figure 7.3: Operation Status Entity Class	69
Figure 7.4: Function Status Entity Class	70
Figure 7.5: Operation Return Type Entity Class	71
Figure 7.6: Successor Operation Entity Class	71
Figure 7.7: Operation Statechart Entity Class	73
Figure 7.8: Operation Statechart Information Entity Class	74
Figure 7.9: Operation Reconfiguration Status Entity Class	75
Figure 7.10: Operation Caller Entity Class	76
Figure 7.11: Operation Callee Entity Class	76

Figure 7.12: Repair Technique Entity Class.....	77
Figure 7.13: Operation Criticality Entity Class.....	79
Figure 7.14: Detection Mechanism Class Diagram.....	81
Figure 7.15: Detection Mechanism Communication Diagram.....	82
Figure 7.16: Reconfiguration Mechanism Class Diagram	86
Figure 7.17: Reconfiguration Mechanism Communication Diagram	87
Figure 7.18: Repair Mechanism Class Diagram.....	91
Figure 7.19: Repair Mechanism Communication Diagram	92
Figure 8.1: Floor COTS Component Software Architecture.....	98
Figure 8.2: Elevator COTS Component Software Architecture	99
Figure 9.1: Receiving Workstation Controller COTS Component Software Architecture.....	142
Figure 9.2: Line Workstation Controller COTS Component Software Architecture.....	144
Figure 9.3: Shipping Workstation Controller COTS Component Software Architecture.....	145
Figure A.1: Algorithms Provided by COTS Modified Interface for Detection Mechanism.....	206
Figure A.2: Algorithms Provided by COTS Monitor for Detection Mechanism.....	206
Figure A.3: Algorithms Provided by Wrapper Controller for Reconfiguration Mechanism.....	209
Figure A.4: Algorithms Provided by Reconfiguration Manager for Reconfiguration Mechanism.....	211
Figure A.5: Algorithms Provided by COTS Modified Interface for Reconfiguration Mechanism.....	215
Figure A.6: Algorithms Provided by COTS Monitor for Reconfiguration Mechanism.....	216

Figure A.7: Algorithms Provided by Repair Manager for Repair Mechanism..	217
Figure A.8: Algorithms Provided by COTS Modified Interface for Repair Mechanism.....	218
Figure A.9: Algorithms Provided by Wrapper Controller for Repair Mechanism.....	219

CHAPTER 1

INTRODUCTION

1.1 Background

Many software systems today are built by incorporating COTS (commercial off-the-shelf) components; rarely are new systems developed from scratch. COTS components are usually designed for general use to be reused in different systems, so they may include many functions for their potential reuse. Although COTS components offer great functionality, they still may not meet user's (or purchaser's) requirements such as self-management (IBM 2003) that autonomously manage abnormal behavior of systems at run-time without human interaction. The properties of self-management are necessary for dependable COTS component-based systems that require high reliability and availability.

Self-management of COTS-component-based systems includes detection of anomalies in COTS components, reconfiguration of anomalous COTS components, and repair of anomalous COTS components, which can be achieved at the level of software architecture by means of self-managed COTS components and interaction between the components. COTS components may need to be changed to support the properties of self-management in a system, but the change to COTS components is not always possible due to COTS vendor's protecting intellectual property. To achieve self-management of a COTS component-based system, self-managed COTS components need to be designed

using given COTS components, without change to the components. In addition, it is necessary to specify a mechanism for the interaction among self-managed COTS components to achieve anomaly detection, dynamic reconfiguration, and repair of anomalous COTS components.

According to a survey of software fault tolerance techniques, self-healing methods, and self-management methods, the approaches have paid less attention to the self-management of COTS component-based systems. In particular, the approaches have less focused on the self-management mechanism for COTS components in terms of detection of anomaly in COTS components, reconfiguration of anomalous COTS components, and repair of anomalies detected. For this, we need to create the self-management mechanism, which consists of the detection mechanism, reconfiguration mechanism, and repair mechanism.

This research approach is applicable for soft real-time COTS component-based systems, which require reliability and availability of services. At the level of the software architecture (Shaw and Garlan 1996; Buschmann et al. 1996) a self-management of COTS component-based system is composed of self-managed COTS components. Each self-managed COTS component is designed using a wrapper (DeLine 1999) which encapsulates the COTS component to control the interaction between COTS components.

1.2 Problem Statement and Research Approach

This research focuses on solving the problem of self-managing COTS component-based systems in a systematic and consistent way. The approach taken to solve the problem includes the following research activities:

- a) Develop a detection mechanism to identify COTS anomalies
- b) Develop a reconfiguration mechanism to minimize the impact from the anomalies detected
- c) Develop a repair mechanism to fix anomalies in the COTS components
- d) Design a wrapper for self-managing COTS components
- e) Validate the detection, reconfiguration and repair mechanism.
- f) Apply the self-managed approach to case studies to validate this research.

The problem statement and research approach are described in more detail in Chapter 3.

1.3 Importance and Rationale of This Research

A self-management of COTS component-based systems approach that autonomously detect and repair at runtime anomalies of the systems (i.e., design faults or unanticipated events) in a systematic and consistent way allows the COTS component to be repaired at runtime without human intervention. Self-managed COTS component are designed using given COTS component and a wrapper, which contain the capability for self-managing the COTS component. The wrapper of a self-managed COTS component detects anomalies in COTS components, reconfigures not only the anomalous COTS components but also the other COTS components of the system, and repairs anomalous COTS components autonomously. The self-management of COTS component-based systems approach provided by this research allows the reusing of a COTS component in different systems and autonomously manages abnormal behavior of systems at runtime.

1.4 Terminology

Important terms used in this dissertation are provided in this section.

COTS Component

A Commercial Off-The-Shelf (COTS) software component is a piece of software sold by a vendor to provide services to a system. Usually COTS component are provided as a black box to be reused ‘as it is’ and its implementation is unknown (or hidden) to the users to protect the vendor’s intellectual property.

Unified Modeling Language

Unified Modeling Language (UML) is a general-purpose visual language used by the software community to formulate a software system specification. UML is designed to be used in object-oriented application development. UML provides not only a standardized notation but also different diagrams, such as use case, class, communication, and statechart diagrams.

1.5 Organization

This dissertation is structured as follows: Chapter 2 provides a survey of related research. Chapter 3 addresses the problem statement and research approach. Chapter 4 describes the detection mechanism to identify COTS anomalies. Chapter 5 develops the reconfiguration mechanism to minimize the impact from the anomalies detected. Chapter 6 describes the repair mechanism to fix anomalies in COTS components. Chapter 7 presents the design of the wrapper for self-managing COTS components. Chapter 8 and 9

illustrate two cases studies: distributed elevator system and distributed factory automation system respectively. Finally, Chapter 10 delineates contributions of this research and further research. At the end, references and appendices are attached.

CHAPTER 2

RELATED RESEARCH

2.1 Introduction

This chapter presents a survey of the current approaches related to the research described in this dissertation and compares those related approaches. This chapter begins by describing autonomic computing in section 2.2. Section 2.3 presents different approaches in the area of software fault-tolerance. Section 2.4 describes mediators and obligations. Section 2.5 presents the idealized C2 component. Section 2.6 introduces protective wrappers detailing several characteristics and roles. Section 2.7 surveys self-healing approaches. Section 2.8 compares all of the approaches discussed in this chapter.

2.2 IBM Autonomic Computing

Autonomic computing (IBM 2003) helps address today's system complexity by using technology to manage technology, anticipating system requirements and resolving problems, with minimal human intervention. In an autonomic computing environment, any autonomic capability is a control loop that gathers details from the system and takes action accordingly. Those control loops can be composed of the same fundamental parts, but their functionality can also be grouped into four embedded categories:

- *Self-configuring*: the system can dynamically adapt to changes in the environment, such as the development of new components, the elimination of existing ones, or changing in the characteristics of the system.
- *Self-healing*: the system can detect malfunctions and begin a policy-based corrective action without disturbing the environment, such as altering the state of their components, or making changes in those system's components.
- *Self-optimizing*: the system can monitor and tune resources automatically, such as reallocating resources to improve overall utilization of the system, or guaranteeing that business transactions can be finished in a timely fashion.
- *Self-protecting*: the system can anticipate, detect, identify and protect itself against threats, enforcing business security and privacy policies.

Autonomic computing describes the autonomic capabilities for an open system architecture for self-managing the systems.

2.3 Software Fault-Tolerance Approaches

Fault-tolerance techniques applicable to software are presented in this section. These techniques are divided into two groups: single-version and multi-version software techniques.

2.3.1 Single-Version Software Fault-Tolerance Techniques

Single-version techniques focus on improving the fault-tolerance of a single piece of software by adding mechanisms into the design targeting the detection, containment,

and handling of errors caused by the activation of design faults (Torres-Pomales 2000).

Single-version software fault-tolerance techniques include:

- *Software structure and actions.* The development of software systems should take into consideration the use of techniques to implement fault tolerance, such as modularizing techniques to decompose a problem into manageable components, and partitioning techniques to provide isolation between functionally independent modules. This development should also use atomic actions and control in order to assure that no action is permissible unless it is explicitly authorized.
- *Error detection.* The structural modules constituting a system have to apply two basic properties: self-protection and self-checking. The self-protection property states that a component must be able to protect itself from external contamination by detecting errors in the information received by other components; while the self-checking property means that a component must be able to detect their own errors and take appropriate action to prevent the error propagation to other components.
- *Exception handling.* This is the interruption of normal operation to handle abnormal responses. When designing a component some considerations have to be given to the design of the exception handlers such as possible events triggering the exceptions, the effect of those events, and the appropriate mitigating action to be selected.
- *Recovery mechanisms.* The most common recovery mechanism is a checkpoint and restart recovery. There are two kinds of restart recovery: static

and dynamic. The static restart is based on returning the module to a predetermined state, while the dynamic restart uses dynamically created checkpoints that are snapshots of the state at various points during the execution, so the module can return to one of those checkpoints dynamically.

- *Process Pairs*. This approach uses two identical versions of the software that run on separate processors. The primary processor runs and generates checkpoint information, which is sent to the backup processor. In the case of error detection, the backup processor loads the last checkpoints and takes over the role of primary processor.
- *Data Diversity*. This is a combination of two techniques: “input sequence workarounds” and checkpoint and restart. Data diversity uses different re-expressions of inputs on each retry increasing the effectiveness of the checkpoint and restart mechanism.

2.3.2 Multi-Version Software Fault-Tolerance Techniques

Multi-version fault-tolerance is based on the use of two or more versions (or variants) of a piece of software, which are executed either in sequence or in parallel (Torres-Pomales 2000).

- *Recovery Blocks*. This combines the basics of the checkpoint and restart approach with multiple versions of a software component. An error is detected when the acceptance test for the corresponding output fails; therefore, a different version is tried. If all the alternates are tried unsuccessfully, the component must raise an exception to the rest of the system its failure.

- *N-Version Programming*. Several versions of software are designed to satisfy the same basic requirements. N-Version Programming differs from Recovery Blocks in the sense that the decision of output correctness is based on the comparison of all the outputs using a generic decision algorithm, which usually is a voter. The execution of the versions can be sequential or in parallel. Sequential execution may require the use of checkpoints to reload the state before an alternate version is executed.
- *N Self-Checking Programming*. This approach is the combination of Recovery Blocks and N-Version Programming, with the difference that the versions and the acceptance tests are developed independently from common requirements. The execution of the versions and their tests can be done sequentially or in parallel, but the output is taken from the highest-ranking version that passes its acceptance test. Sequential execution requires the use of checkpoints, and parallel execution requires the use of input and state consistency algorithms.
- *Consensus Recovery Blocks*. This approach combines N-Version Programming and Recovery Blocks to improve reliability of software. Consensus Recovery Blocks uses a decision algorithm (similar to N-Version Programming) as a first layer of decision. If the decision algorithm declares a failure, an acceptance test (similar to Recovery Blocks) is invoked. Although this approach is more complex than either of the individual techniques, the reliability models indicate that this combined approach has the potential of producing a more reliable piece of software.

- *t/(n-1)-Variant Programming*. This approach is different from the ones described above in the mechanism used to select the output from multiple variants. The design of the selection logic is based on the theory of system-level fault diagnosis, which is beyond the scope of this research. Basically, a *t/(n-1)-Variant Programming* architecture guarantees that at least one non-faulty unit exists such that its output is correct and can be used as the result of the computation for the module.
- *PSRR Scheme*. This approach presents an object-based, real-time fault tolerance scheme called the *primary-shadow RTO.k replication* (PSRR) scheme (Kim and Subbaraman 1997). The structure of an RTO.k object incorporates two extensions to the conventional object model: a) spontaneous methods, called time-triggered methods, clearly separated from the conventional service methods triggered by messages from clients; and b) a deadline that is imposed for each execution of a method of an RTO.k object. Each object in the system is replicated to form a pair of partner objects, and the partner objects are executed in two different nodes; node A hosts the initial primary RTO.k object and node B hosts the initial shadows RTO.k object. The primary and the shadow process the client request (service method), concurrently saving the client request message into the log cache, and invoke their self-checking independently and concurrently by using the same acceptance test (AT) routine. When the primary service method fails its AT, it informs its shadow partner of the failure and changes its role to that of shadow. As a new shadow (in recovery mode), it rolls back to the beginning

and retries processing the client request saved in the log cache. The shadow service method execution changes its role to that of the primary and performs the appropriate output actions (forward recovery). The shadow service method execution will take the roll of the primary service method when there is an absence of an AT result notice from the primary within a specific deadline, or when there is an absence of an output success notice from the primary within a specific deadline.

- *DiRTISs Approach.* As an extension of the previous research, the University of California at Irvine presents a new approach for distributed real-time information systems (DiRTISs) (Kim and Subbaraman 1999). In this approach, if a node becomes unusable, then it must be detected quickly and the software objects hosted in that node must be reactivated on other healthy node(s). This dynamic reconfiguration management (DRM) must be done in a timely fashion without violating the service interruption bound requirement. This approach uses the time-triggered message triggered object (TMO), formerly called the RTO.k object. DCM is a distributed software subsystem, and it is structured in three layers: a) network surveillance (NS) subsystem, b) network reconfiguration (NR) subsystem, and c) object distribution (OD) subsystem. The NS subsystem enables fast learning by each interested fault-free node in the system of the faults or repair-completion events occurring in other parts of the system. Once there is a need for a change in the network configuration such as a node or a link crash, or a new node is incorporated to the system, the NS subsystem detects this change, and then establishes a network of healthy

nodes and healthy links. Whenever the network configuration is changed by the NR subsystem and new nodes are added, the OD subsystem establishes a new distribution of TMOs among the healthy nodes.

2.4 Mediators and Obligations

This approach is proposed by the Honeywell Technology Center, Minneapolis and is designed for soft real-time systems that include some COTS components with client/server architectures (Thomas et al. 2000). Mediators are software entities that detect faults by monitoring interactions between client and server components, enforcing client and server obligations. Faults occur in the system when one or more components fail to meet their obligations, such as timely response to service requests, correctness of results, legitimacy of interactions, and/or unauthorized or illegal interaction between components.

Client and server obligations are enforced by client-side and server-side mediators respectively. Requests from the client and the responses to these requests are monitored by the client-side mediators, while the incoming service requests and outgoing responses are monitored by the server-side mediators. Mediators can provide multiple functions including:

- *Enforcing timing obligations.* Timing obligations of servers ensure that clients do not block for an unpredictable amount of time waiting for a request to be serviced.

- *Fault-management.* To increase system dependability fault-management techniques such as server replication and automatic restart of failed services are used.
- *Security.* Encryption and other security mechanisms are often used to implement policies that determine which clients can access particular services and how secure these interactions must be.
- *Flow control.* This is a policy designed to provide all components with fair access to services.
- *Acceptance test for inputs and outputs.* This ensures that clients do not make erroneous or ill-formed service requests and servers do not return erroneous or ill-formed responses.
- *Controlled introduction of new software.* A new version of software goes through a verification period where they execute in parallel with the old version. Results produced by the new version are compared with results produced by the old version, and in case of disagreement, results from the old version are used.
- *System monitoring/instrumentation.* This is a technique used to identify performance bottlenecks and to detect abnormal interaction patterns among components.

2.5 Idealized C2 Component (iC2C)

This approach is proposed by the Universidade Estadual de Campinas, Brazil. An idealized C2 Component (iC2C) is a component whose structure and behavior is

equivalent to the idealized fault-tolerant component. This is designed to be used in a C2 architecture style (Guerra, Rubira, and de Lemos 2002). An idealized fault-tolerant component (iFTC) is structured internally into two distinct parts: one that implements its normal behavior and another that implements its abnormal behavior, which deals with the exceptional conditions. The communication among iFTCs is only through request/responses messages. When a request of service is received, an iFTC will react with a normal response if the request is successfully processed. External exceptions may be occurred due to an invalid service request (interface exception), or due to a failure in processing a valid request (failure exception).

The normal activity and abnormal activity parts of an iFTC are decoupled to minimize the impact of fault tolerance provisions on the system complexity, leading to an overall structure for the iC2C consisting of two distinct components (normal and abnormal) and three connectors (top, bottom, and internal). The functionality of an iC2C's normal and abnormal components is equivalent to the iFTC's. The top connector connects the iC2C with the upper components of a C2 configuration, while the bottom connector connects the iC2C with the lower components of a C2 configuration. The internal connector controls the message flow inside the iC2C, selecting the destination of each message received based on its originator, the message type, and the operational state of the iC2C.

2.6 Protective Wrappers

A wrapper is a specialized component inserted between a component and its own environment to deal with the flows of control and data going to and/or from the wrapped

component. Wrappers can be employed for improving quality properties of the components such as adding caching and buffering, dealing with mismatches or simplifying the component interface (Guerra et al. 2004). The City University at London (Meulen et al. 2005) has proposed several characteristics and roles for protective wrappers:

- Wrapping has the advantage of not requiring detailed knowledge of the internal structure of the component being wrapped; therefore it is suitable for integrating COTS components.
- The expected or required behavior of the components needs to be described in some unambiguous language, such as preconditions and postconditions.
- The specification for the wrapper may differ from the model of the COTS component, even in its normal behavior, by hiding from the rest of the system some of the functions offered by the COTS component, and by adding dependability requirements, which determine the need for fault-tolerance provisions in the wrapper.
- A COTS component often precludes close monitoring for early error detection, so wrappers need to react to a pattern of component behavior that merely suggests a failure.
- Wrappers may provide an error recovery mechanism, such as restore the component to an initial state, so the COTS component will subsequently exhibit correct behavior, or if a COTS component has an ‘undo’ operation, the wrapper could use it for backwards recovery and retry.

The Universidade Estadual de Campinas, Brazil presents an approach that combines the concepts of an idealized C2 component (iC2C) and protective wrappers to develop an architectural solution for building dependable software systems from Idealized C2 COTS (iCOTS) component (Guerra et al. 2003; Guerra et al. 2004). An iCOTS component is a specialization of the iC2C that employs protective wrappers for encapsulating a COTS component. The COTS component is connected to two specialized connectors (lower_detector and upper_detector) acting as error detectors to compose the normal activity component of the iCOTS component. The lower_detector connector inspects incoming requests and outgoing responses from/to the COTS clients, while the upper_detector connector inspects outgoing requests and incoming responses to/from other components. Both detectors are responsible for verifying the messages that flow to/from the COTS component being wrapped do not violate the acceptable behavior constraints specified for the system.

The abnormal activity component of an iCOTS component is responsible for both error diagnosis and error recovery. Depending on the complexity, the abnormal activity component can be decomposed into more specialized components for error diagnosis and a set of error handlers. The error diagnosis component is able to react directly to exceptions raised by the normal activity component and send notifications to the error handlers.

2.7 Self-Healing Approaches

2.7.1 Elements of the Problem Space

Intuitively a self-healing system is able to ‘heal’ itself. There are four general categories of aspects to the self-healing system problem space (Koopman 2003):

- *Fault Model.* Self-healing systems have a fault model in terms of what faults they are expected to be able to self-heal. A fault model takes into consideration the following: the duration of the fault (permanent, intermittent, or transient) to understand what situations it addresses; the manifestation of the fault because not all faults are as severe as others; the source of the fault because it can affect the self-healing strategies; the granularity of the fault, understanding granularity as the size of the component that is compromised by the fault; and the profile of fault occurrences that is expected.
- *System Response.* The first step responding to a fault is actually detecting the fault. The extra-functional concerns that influence the system responses are as follows: fault detection, which can be performed internally by a component, by comparing replicated components, by peer-to-peer checking, or by supervisory checks; degradation, a self-healing system might not restore complete functionality after a fault; fault response, once a fault has been detected, the system must select a response mechanism (masking a fault, rollback, rollforward, or retrying using alternate resources); recovery, after a system has detected a fault and invoked a fault response, the system must recover the faulty operation to complete the self-healing process; time

constants, along with the fault distribution assumptions, determine what types of self-healing are feasible; and assurance, self-healing systems require assurances that some level of functional and extra-functional correctness for normal operation are maintained during and after fault occurrences.

- *System Completeness.* Self-healing approaches must be able to deal with the reality of today's real systems. This is because few system architectures are completely elaborated when the first implementation is built, designers do not have complete knowledge of the systems they design, systems change over time, or systems has incomplete specifications and designs.
- *Design Context.* The design context of the system can be formed by several other factors that influence the scope of self-healing capabilities: abstraction level -application software, middleware mechanisms, operating systems, hardware; component homogeneity - while some systems have completely homogeneous components, it is common to have systems that are heterogeneous to some degree; behavioral predetermination - most systems do not have perfectly predetermined and deterministic behavior; user involvement in healing - limited healing ability beyond which users must become involved in system repair; system linearity and component coupling can affect the ability of a system to self-heal; and finally, the system scope.

2.7.2 Architecture-based Approaches

One or more models of a system can be maintained at run time and are external to the application as a basis for identifying problems and resolving them (Garlan and

Schmerl 2002; Garlan, Cheng, and Schmerl 2003). This approach is attractive for different reasons. First, depending on the system quality of interest, different models can be chosen. Second, reuse of externalized mechanisms is possible since they are not application-specific. Third, they can be easily changed, since they are localized. Forth, they can exploit a large body of exiting work on analytical methods for improving attributes such as performance, reliability, or security.

The centerpiece of this approach is the use of architectural models which are represented as a graph of interacting components and connectors, and which can also specify a set of constraints that must be maintained. An executing system is monitored to observe its run time behavior. Monitored values are abstracted and related to architectural properties of an architectural model. If there is a change in the properties of the architectural model, a constraint evaluation is triggered. The analyzer determines whether the system is operating within an envelope of acceptable ranges. A repair mechanism handles those constraint violations and adapts the architecture. Architectural changes are propagated to the running system using a translator.

Another approach to creating self-healing systems based on software architecture is presented by the University of California at Irvine where repairs are done at the level of a software system's components and connectors (Dashofy, Hoek, and Taylor 2002). This approach is applied to event-based architectures, where components are not allowed to directly communicate with each other; rather they communicate via their connectors. Therefore, components are essentially ignorant of their location in the architectural topology allowing them to be inserted, removed and replaced without explicit changes to their code. In this approach, the ability to dynamically repair a system at runtime based

on its architecture requires several capabilities: the ability to describe the current architecture of the system; the ability to express an arbitrary change to that architecture that will serve as a repair plan; the ability to analyze the result of the repair to gain confidence that the change is valid; and the ability to execute the repair plan on a running system without restarting the system. The architectures of the software systems are described in xADL 2.0 (XML-based architecture description language) containing information about the location and format of component and connector implementations.

An architectural repair can be expressed in terms of an architectural difference, which is a document that describes the differences between two software architectures (before a repair and after a repair). The architectural difference schema consists of two sequences: one of additions and one of removals. Elements in the architectural difference include components, connectors, links, component types, connector types, and interface types. Differences are primarily determined by the presence of new elements or the absence of old elements in the ‘after’ description, but changes to existing elements are also detected by comparing the contents of element descriptions. The architecture analysis is performed by design critics. Each design critic monitors open architecture descriptions in the architecture for changes. When a change is made, the design critics check the description for any problematic issues, and then they report to a central issue database. If all appropriate critics finish their analysis and have not reported any issues with the changed architecture description, then a repair planning agent can have confidence that making the change to the running system will be successful; therefore, the repair plan is executed.

2.8 Comparison Among the Approaches

The following Table 2-1 shows the comparison among the approaches surveyed. The comparison criteria include the type of the system and the architecture where each approach can be applied, and what the mechanisms each approach used for detecting, reconfiguring and repairing the corresponding system.

After surviving the different approaches, a general conclusion can be deduced: no one of the approaches autonomously manage abnormal behavior of systems at run time without human interaction. The approaches surveyed use acceptance test, decision or selection algorithms, or pre and post conditions as mechanisms to detect a fault. The problem with those mechanisms is that they depend on the component that are monitoring, and in some cases they need other versions of the same component to work successfully. The reconfiguration mechanism that the approaches surveyed propose take into consideration the faulty component but they do not specify a mechanism to reconfigure the rest of the system during the reconfiguration of the faulty component. The repair mechanisms proposed by those approaches only consider roll back in the form of checkpoints or backward recovery. Again, they do not specify what happen to the rest of the system during the execution of the repair mechanism.

Many software systems today are built by incorporating COTS components; rarely are new system developed from scratch. There are only two approaches that deals with COTS components, but those approaches do not manage autonomously the abnormal behavior of system at run-time without human intervention.

Table 2.1: Comparison Among the Approaches Surveyed

Criteria Method	Type of System	Architecture	Fault-detection Mechanisms	Reconfiguration Mechanisms	Repair Mechanisms
Single-Version Software Fault- Tolerance	All systems	Components	<ul style="list-style-type: none"> Self-protection Self-checking 	Exception handling	<ul style="list-style-type: none"> Checkpoint/Restart Data re-expression
Multi-Version Software Fault-Tolerance:					
<ul style="list-style-type: none"> Recovery Blocks 	All systems	+2 Component versions	Acceptance test	Exception handling	Checkpoint/Restart
<ul style="list-style-type: none"> N-Version Programming 	All systems	+2 Component versions	Decision algorithm	Exception handling	Checkpoint/Restart
<ul style="list-style-type: none"> N-Self-Checking Programming 	All systems	+2 Component versions	Acceptance tests	Exception handling	Checkpoint/Restart
<ul style="list-style-type: none"> Consensus Recovery Blocks 	All systems	+2 Component versions	<ul style="list-style-type: none"> Decision algorithm Acceptance test 	Exception handling	Checkpoint/Restart
<ul style="list-style-type: none"> t(n-1)-Variant Programming 	All systems	+2 Component versions	Selection algorithm	N/A	N/A
<ul style="list-style-type: none"> PSRR 	OO Real-Time Systems	Primary/Shadow objects	Acceptance test	NSR manager	Roll back
<ul style="list-style-type: none"> DiRTISs 	Distributed RT Systems	Nodes	Acceptance test	DR manager	Roll back
Mediators/Obligations	Soft RT Systems	Client/Server	Mediators	Mediators	Not specified
iC2C	All systems	C2 architecture style	<ul style="list-style-type: none"> Pre/Post Conditions Invariants 	Abnormal activity component	Not specified
Protective Wrappers:					
<ul style="list-style-type: none"> City University 	All systems(COTS)	COTS wrapped	Pre/Post Conditions	Replacement with standard value	Backward recovery
<ul style="list-style-type: none"> iCOTS 	All systems(COTS)	COTS wrapped (C2 architecture style)	Acceptable behavior constraints (ABCs)	Exception handling	Not specified
Self-healing:					
<ul style="list-style-type: none"> Carnegie Mellon University 	All systems	Any	Changes on the architectural model	Adapts the architecture	Adapts the architecture
<ul style="list-style-type: none"> University of California 	All systems	Components and connectors	Design critics	Architectural difference documents	Architectural difference documents

CHAPTER 3

PROBLEM STATEMENT AND RESEARCH APPROACH

3.1 Introduction

The purpose of this research is to develop an approach to self-managing COTS component-based systems using wrappers in a systematic and consistent way. In order to develop the self-managing approach, this research develops a detection mechanism to identify anomalies in the COTS components; a reconfiguration mechanism to minimize the impact from the anomaly detected; and a repair mechanism to fix the anomaly in the anomalous COTS component. This research designs the wrapper to support the self-managing mechanisms. The self-managing approach for the COTS component-based system is validated through two case studies.

This chapter addresses the problem statement for this research and the research approach to solve the problem. The chapter begins with the problem statement for this research in section 3.2. Section 3.3 describes the research approach. Section 3.4 summarizes this chapter.

3.2 Problem Statement

This research solves the following problem:

Current approaches address self-management of COTS component-based systems in an ad-hoc way. There is no self-management mechanism that addresses self-

management of COTS component-based systems in a systematic way. Self-managed COTS components need to be capable of detecting anomalies in COTS components, reconfiguring COTS components against the anomalies, and repairing the anomalies in COTS components.

3.3 Research Approach

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

- a) Develop a detection mechanism to identify COTS component anomalies. A COTS component may contain malfunctions that can be caused by such anomalies in the component as design errors, unfair resource allocation, or inappropriate controlling of concurrency among threads in the component.
- b) Develop a reconfiguration mechanism to minimize the impact of the anomalies detected. An anomalous function provided by a COTS component may need to be isolated from other healthy COTS components in the system so that a system reduces the impact from the anomalous function or COTS component. As an anomalous COTS component resumes its services after repair, the COTS component also need to reconfigure the system so as to provide full functionality of the system.
- c) Develop a repair mechanism to fix anomalies in the anomalous COTS component. Some anomalies in a COTS component are critical to the whole system, which may lead to system failures. The repair of anomalies in COTS component is to protect the system from system failures.

- d) Design of a wrapper for self-managing COTS component-based systems. A self-managed COTS component is designed using a wrapper (DeLine 1999), which encapsulates the COTS component to control the interaction between COTS components.
- e) Finally, apply the self-managing approach to two case studies to validate this research. Two case studies include a distributed elevator system and a distributed factory automation system.

3.4 Summary

This chapter has addressed the problem statement for this research and the research approach.

CHAPTER 4

DETECTION MECHANISM TO DETECT COTS COMPONENTS ANOMALIES

4.1 Introduction

This chapter develops a mechanism capable of monitoring a COTS component to detect anomalies that may result in system failures. A COTS component may contain malfunctions that can be caused by such anomalies in the component as design errors, unfair resource allocation, or inappropriate controlling of concurrency among threads in the component. These anomalies within a COTS component can make the systems unreliable and unavailable.

This chapter starts with the software architecture for self-managed COTS component-based systems in Section 4.2. Section 4.3 describes the detection mechanism. Section 4.4 presents the high-level view of algorithm for the detection mechanism. Section 4.5 summarizes this chapter.

4.2 Software Architecture for Self-Managed COTS Component-Based Systems

The software architecture (Shaw and Garlan 1996; Buschmann et al. 1996) for making COTS component-based system dependable is designed by means of self-managed COTS components and interaction between the components. A self-managed COTS component is capable of autonomously managing anomalies attributed to design faults in the component and adapting itself in response to changes in other components. A

COTS component is anomalous if the component does not behave as specified. Self-managed COTS components interact with each other to self-manage anomalies detected as well as provide functional services.

Self-managed COTS components are designed using wrappers (DeLine 1999), which contain the capability for self-managing the COTS components. A self-managed COTS component provides services to and/or requests them from other components via its wrapper, whereas a wrapper offers an interface to its COTS component and to other components. The wrapper intercepts the message communication between the COTS component being wrapped and other COTS components.

In designing self-managed COTS components, each COTS component is used as sold by COTS component vendors. That is, a COTS component is considered a black box whose internal details are hidden except for its original interface not only to services the component provides but also to the services the component requires.

The software architecture consisting of self-managed COTS components with wrappers is depicted in Figure 4.1, in which each COTS component communicates with others via its wrapper. The COTS component *COTS C1* is self-managed by the COTS component wrapper *COTS C1 Wrapper*. The COTS component *COTS C1* receives a service request from the COTS component *COTS C2* via the COTS component wrapper *COTS C1 Wrapper* and the COTS component wrapper *COTS C2 Wrapper*. The response from the COTS component *COTS C1* is also delivered to the COTS component *COTS C2* via the COTS component wrapper *COTS C1 Wrapper* and the COTS component wrapper *COTS C2 Wrapper*. Similarly, the COTS component *COTS C2* communicates with the COTS component *COTS C3* via the COTS component wrappers *COTS C2 Wrapper* and

COTS C3 Wrapper. The communication between COTS components *COTS C3* and *COTS C1* is performed via the COTS component wrappers *COTS C3 Wrapper* and *COTS C1 Wrapper*.

Figure 4.1: Software Architecture with Self-Managed COTS Components

4.3 Detection Mechanism

The functions provided by a COTS component are monitored by the *COTS component wrapper*. The *COTS component wrapper* encapsulates statecharts (Booch, Rumbaugh, and Jacobson 2005; Rumbaugh, Booch, and Jacobson 2005) that describe the states of each function provided by a COTS component. The statechart for a function in a COTS component models the states the function can stay. When a function in a COTS component is invoked, the statechart for the function in the *COTS component wrapper* makes transition from one state to the next state. Some function in a COTS component may need to invoke other functions in a different COTS component to perform its own

service. The statechart describes states and transitions related to this interaction between components. The statechart for a function models states and their transitions from the beginning of the function to the completion.

The statecharts for each function encapsulated in the *COTS component wrapper* are executed by the messages invoked by the clients of the component, such as different COTS components or external objects. That is, the *COTS component wrapper* is an intermediary in the interaction between a COTS component and its clients. The *COTS component wrapper* monitors the messages passing between a COTS component and its clients when they communicate with each other. The messages are used to execute the statechart in the *COTS component wrapper* to determine when and which function in a COTS component becomes anomalous.

The *COTS component wrapper* maintains the status of each function in a COTS component, describing whether a function performs its activity as specified. The status of a function in a COTS component is ‘unblocked’ if the function performs its obligation normally. On the other hand, the status of a function is marked ‘blocked’ if the function contains its own anomalies (referred to as ‘malfunction block’) or the function requires other function in a different COTS component that is anomalous (referred to as ‘dependency block’ or ‘function block’). In addition, the status of a function can be blocked during the repair of an anomalous COTS component (referred to as ‘repair block’). When a client requests a function in a COTS component, the *COTS component wrapper* checks the status of the function requested. Only functions whose status is ‘unblocked’ are allowed to be called by the *COTS component wrapper*.

Figure 4.2 depicts the self-management of COTS components of part of the Distributed Elevator System (Gomaa 2000) with multiple elevators using the collaboration diagram of UML (Booch, Rumbaugh, and Jacobson 2005; Rumbaugh, Booch, and Jacobson 2005). The message sequence A1 through A4 describes the monitoring of the Floor Button Request function provided by the Floor COTS component, which is requested by an elevator user when he or she presses the floor button at a specific floor through a Floor Button input device. When a passenger requests an elevator, this request is delivered to the Scheduler COTS component, which adds the request to the scheduler plan, and determines what elevator is going to the elevator user. The message sequence A5 through A7 in Figure 4.2 describes the monitoring of the Service Request function provided by the Scheduler COTS component.

Figure 4.2: Floor and Scheduler COTS Component of the Elevator System

When the *Floor Button* input device sends an elevator request (message A1) to the *Floor COTS Component* through the *Floor Wrapper*, the *Floor Wrapper* checks the

status of **Floor Button Request** function in the *Floor COTS Component*. If the status of the **Floor Button Request** is blocked, this function is not allowed to be called by the input device. However, if the status of the **Floor Button Request** function is unblocked, the statechart for the floor button request function (Figure 4.3) encapsulated in the *Floor Wrapper* makes transitions from the ‘Idle’ state to the ‘Calling Floor Button Request’ state by the “Floor Button Request Arrived” notification message. The *Floor Wrapper* then calls the **Floor Button Request** function (message A2 in Figure 4.2) in the *Floor COTS Component*. The statechart for the floor button request function in the *Floor Wrapper* makes transition from ‘Calling Floor Button Request’ state to the ‘Processing Floor Button Request’ state by the “Floor Button Request Called” notification message. Similarly, when the *Floor COTS component* finishes processing the floor button request function, the *Floor COTS Component* sends the **Require Service Request** message (message A3 in Figure 4.2) to the *Floor Wrapper*. The statechart for the floor button request function makes a transition from the ‘Processing Floor Button Request’ state to the ‘Calling Service Request’ state by the “Service Request Arrived” notification message. The *Floor Wrapper* then sends the **Service Request** function (message A4 in Figure 4.2) to the *Scheduler Wrapper*, so the *Scheduler COTS Component* adds the floor and direction in the Scheduler’s plan. The statechart for the floor button request function encapsulated in the *Floor Wrapper* makes transition from the ‘Calling Service Request’ state to the ‘Idle’ state by the “Service Request Called” notification message.

Figure 4.3: Statechart for Floor Button Request Function

When the *Scheduler Wrapper* receives the **Service Request** message (message A4 in Figure 4.2) from the *Floor Wrapper*, the *Scheduler Wrapper* checks the status of the **Service Request** function in the *Scheduler COTS component*. If the status of the **Service Request** function is unblock, the statechart for the service request function (Figure 4.4) encapsulated in the *Scheduler Wrapper* makes a transition from the 'Idle' state to the 'Calling Service Request' state by the "Service Request Arrived" notification message. The *Scheduler Wrapper* then calls the **Service Request** function (message A5 in Figure 4.2) in the *Scheduler COTS Component*, and the statechart for the service request function makes a transition from the 'Calling Service Request' state to the

‘Processing Service Request’ by the “Service Request Called” notification message.

When the *Scheduler COTS Component* finished adding the floor and direction to the Scheduler’s plan, and selecting an elevator to be sent to an elevator user, the *Scheduler COTS Component* sends the **Schedule Request** message (message A6 in Figure 4.2) to the *Scheduler Wrapper*. The statechart for the schedule request function encapsulated in the *Scheduler Wrapper* makes a transition from the ‘Processing Service Request’ state to the ‘Calling Schedule Request’ state by the “Service Request Arrived” notification message. The *Scheduler Wrapper* then calls the **Schedule Request** function (message A7 in Figure 4.2) in the *Elevator Wrapper* selected. The statechart for the service request function makes a transition from the ‘Calling Schedule Request’ state to the ‘Idle’ state by the “Schedule Request Called” notification message.

Figure 4.4: Statechart for Service Request Function

The *Floor Wrapper* and *Scheduler Wrapper* presume that a function provided by the *Floor COTS Component* and the *Scheduler COTS Component* respectively may be anomalous if the expected notification messages (messages A3 and A6 in Figure 4.2) have not arrived within reasonable time intervals. Each state in the statecharts encapsulated in the *Floor Wrapper* and *Scheduler Wrapper* should make transitions within expected time intervals once the statechart starts transitions. Otherwise, the *Floor Wrapper* and the *Scheduler Wrapper* report the anomaly and take an appropriate action against the anomalous function.

4.4 High-Level View of Algorithm for Detection Mechanism

A high-level view of the algorithm for the detection mechanism can be described as follows:

- 1: An external user or other COTS component requests a service from the *COTS component wrapper*.
- 2: The *COTS component wrapper* receives the service request identifying the function name and parameters.
- 3: The *COTS component wrapper* checks whether the status of the function is ‘unblock’. If the status of the function is ‘malfunction block’, ‘dependency block’, or ‘repair block’, the *COTS component wrapper* does not allow this function to be invoked. If the status of the function is ‘unblock’, the *COTS component wrapper* starts monitoring the function and invokes the function from the COTS component.

- 4: The statechart for the function makes transition from the 'Idle' state to the next state and the *COTS component wrapper* starts timing the execution of the function.
- 5: The COTS component performs the function requested by the *COTS component wrapper*.
- 6: When the COTS component finishes processing the function, the COTS component sends the response back to the *COTS component wrapper*.
- 7: The *COTS component wrapper* receives the response from the COTS component. The *COTS component wrapper* stops timing the function and calculates whether the corresponding execution time is less or equal to the maximum execution time established for this function. If the execution time is between boundaries, the statechart makes a transition to the next corresponding state.
- 9: The *COTS component wrapper* identifies the next function request to be invoked. The statechart makes transitions again to the next state.
- 10: If the maximum execution time for a function has elapsed, and the *COTS component wrapper* does not receive a response from the COTS component, then the *COTS component wrapper* suspects that the function is not working properly, and generates a malfunction notification. This malfunction notification is the end of the detection mechanism and the start of the reconfiguration mechanism is described in chapter 5.

4.5 Summary

This chapter has described the software architecture for making COTS component-based systems dependable by means of self-managed COTS component and the interaction between the components. Each COTS component has been encapsulated in a wrapper that is capable of detecting anomalies in the COTS component. The *COTS component wrapper* monitors the functions provided by a COTS component using statecharts. The statechart makes transition from one state to the other state based on the message received by the *COTS component wrapper*. If the message communications does not arrive within a reasonable time interval, the *COTS component wrapper* has detected the anomaly of the function and has taken an appropriate action. To understand how this mechanism works the high-level view of the detection mechanism has been described.

CHAPTER 5

RECONFIGURATION MECHANISM TO MINIMIZE THE IMPACT FROM THE ANOMALIES DETECTED

5.1 Introduction

This chapter develops a mechanism that is capable of reconfiguring a COTS component-based system at runtime so as to minimize the impact from the anomalies detected. An anomalous function provided by a COTS component or anomalous COTS component itself may need to be isolated from other healthy COTS components in the system so that a system reduces the impact from the anomalous function or COTS component. As an anomalous COTS component resumes after repair, *COTS component wrappers* also need to reconfigure the system so as to provide full functionality of the system.

The reconfiguration of COTS component-based systems is performed by the *COTS component wrappers*, which have detailed information associated with the current configuration of COTS components. The information in the *COTS component wrappers* includes the status of each function in a COTS component, and reconfiguration plans against anomalous functions within a COTS component and in different COTS components. *COTS component wrappers* co-operate with each other to reconfigure COTS components at runtime, so that the neighboring COTS components have the minimal impact from anomalous COTS components.

This chapter starts with describing the reconfiguration mechanism in section 5.2. Section 5.3 describes a high-level view of the algorithm of the reconfiguration mechanism. Section 5.4 summarizes this chapter.

5.2 Reconfiguration Mechanism

The reconfiguration of a self-managed COTS component is performed by the *COTS component wrapper*. A *COTS component wrapper* contains information about the current configuration of the COTS component, that is, what functions provided by a COTS component are blocked or unblocked due to anomalies within the COTS component and in different COTS components. A *COTS component wrapper* also has information about functional dependencies between each function in the component and functions in different components. Using these dependency relationships, the *COTS component wrapper* generates reconfigurations plans against anomalies within the COTS component and in different COTS components, performing the plans to reconfigure the COTS component by blocking or unblocking functions associated with anomalies.

COTS component-based systems are reconfigured by means of blocking or unblocking the interfaces to each function of COTS components encapsulated in the *COTS component wrapper*. Anomalous functions are blocked so that other components cannot request the function paralyzed. A function can be blocked when the *COTS component wrapper* determines that the function has anomalies, referred to as ‘malfunction block’ (described in chapter 4). A function is also blocked when its required functions in different COTS components cannot provide services normally, referred to as ‘dependency block’. A function is also blocked during the repair process, referred to as

‘repair block’. The status of a function is changed into ‘unblocked’ as the function resumes its service normally. The *COTS component wrapper* always maintains the status of each function of a COTS component.

A *COTS component wrapper* determines whether a function in a COTS component should be repaired immediately or not. The *COTS component wrapper* maintains information about function type such as critical or non-critical functions. With the notification messages about anomalous functions from the detection mechanism (chapter 4), the *COTS component wrapper* makes decisions as to what actions will be taken against the anomalous functions – either just blocking anomalous functions (referred to as ‘malfunction block’), or both blocking and repairing the whole COTS component encapsulating the anomalous function immediately (referred to as ‘repair block’).

Based on the criticality of the functions provided by a COTS component, a malfunction function could be blocked in two different ways. A blocking is ‘repair block’ if a function is critical to the system, whereas a blocking is ‘malfunction block’ if the function is not critical to the system. Both types of blockings may cause other COTS components to block, referred to as dependency block.

The reconfiguration process is carried out by reconfiguration plans. The *COTS component wrapper* generates those reconfiguration plans using the current configuration of the COTS components and the functional dependencies among COTS components. Each reconfiguration plan contains the name of the function to be reconfigured, the action to be performed on the functions and on the dependent functions (‘repair block’,

‘malfunction block’, ‘dependency block’, ‘function block’, and ‘unblock’) and what component provides such functions.

5.2.1 Reconfiguration of Critical Functions

When a critical function in a COTS component has an anomaly, the COTS component providing that function should be blocked for repairing. The *COTS component wrapper* of the COTS component generates a reconfiguration plan for this faulty critical function. The reconfiguration plan needs to block all the functions provided by the faulty COTS component as ‘repair block’. Using the functional dependencies information associated with each function, the *COTS component wrapper* of the faulty COTS component sends notification messages to the *COTS component wrappers* of the dependent COTS components so that they can reconfigure themselves in response to the notification message. The functions in the dependent COTS components that depend on the anomalous function are blocked as dependency block. With these notification messages, the faulty COTS component is isolated from the dependent COTS components. When the notification arrives at the dependent COTS components, each *COTS component wrapper* generates a reconfiguration plan against the repair of the malfunction.

The Figure 5.1 depicts the reconfiguration of a critical function of the distributed elevator system (Gomaa 2000) with multiple elevators using the collaboration diagram of UML (Booch, Rumbaugh, and Jacobson 2005; Rumbaugh, Booch, and Jacobson 2005).

Figure 5.1: Reconfiguration for a Critical Function of the Elevator System

When the *Scheduler Wrapper* detects a malfunction in the **Service Request** function (chapter 4), the *Scheduler Wrapper* checks the criticality of the **Service Request** function to generate the corresponding reconfiguration plan. Because the **Service Request** function is a critical function for the Elevator System, the *Scheduler COTS Component* should be repaired immediately. Although the repair of the anomalous *Scheduler COTS Component* may result in inconsistency of component states, the *Scheduler COTS Component* should be repaired immediately. It is because there is no way to prevent system failure from the malfunction of the function. To generate the

reconfiguration plan, the *Scheduler Wrapper* contains the functional dependencies between each function in the *Scheduler COTS Component* and the functions in different COTS components. In these functional dependencies, the *Floor COTS Component* and the *Elevator COTS Component* have functions that are dependent on the functions provided by the *Scheduler COTS Component*. The *Scheduler COTS Component* must be isolated for the rest of the system before the repairing process takes place. To isolate the *Scheduler COTS Component* from the rest of the Elevator System, the *Scheduler Wrapper* sends a notification to the *Floor Wrapper* (message B1) and to the *Elevator Wrapper* (message B1a) saying that the *Scheduler COTS Component* has to be repaired. When the *Scheduler Wrapper* receives the notification from the *Floor Wrapper* (message B2) and from the *Elevator Wrapper* (message B2a) indicating that those COTS component have been isolated and reconfigured, the *Scheduler Wrapper* proceeds to reconfigure the *Scheduler COTS Component* blocking all functions as ‘repair block’.

After the *Scheduler COTS Component* has been repaired (chapter 6), the *Scheduler Wrapper* unblocks all its own functions and notifies to both the *Floor Wrapper* (message B3) and the *Elevator Wrapper* (message B3a). The *Floor Wrapper* (message B4) and the *Elevator Wrapper* (message B4a) send a message to the *Scheduler Wrapper*, which indicates that the reconfigurations have been executed and the dependent functions have been unblocked.

The *Scheduler Wrapper* maintains the functional dependency relationship between each function in the *Scheduler COTS Component* and functions in the dependent COTS component (i.e., Floor COTS Component and Elevator COTS Component) in the Scheduler Reconfiguration Table (Table 5.1). For each function provided by the

Scheduler COTS Component, the Schedule Reconfiguration Table specifies callee and caller functions and components. The Service Request function in the Scheduler COTS component is called by the Floor Button Request function in the Floor COTS component (caller). The Service Request function in the Scheduler COTS component calls the Schedule Request function in the Elevator COTS component (callee).

Table 5.1: Scheduler Reconfiguration Table

Function	Callee			Caller	
	Function	Function Block	Component	Function	Component
Service Request	Schedule Request		Elevator	Floor Button Request	Floor
Update Status Plan				Schedule Request	Elevator
				Elevator Button Request	Elevator
Update Elevator Status				Sensor Input	Elevator

The reconfiguration plan for the critical **Service Request** function in the Scheduler Wrapper is generated using the Scheduler Reconfiguration Table 5.1. All the functions provided by the *Scheduler COTS Component* have to be blocked as ‘repair block’. If a function has caller functions in the reconfiguration table, the reconfiguration plan includes caller components so that the caller components should be notified for dependency block. The **Service Request** function has the *Floor COTS Component* in caller, whereas **Update Status Plan** and **Update Elevator Status** functions have the *Elevator COTS Component* in caller. Therefore, the *Scheduler Wrapper* has to notify to the *Floor Wrapper* (message B1 in Figure 5.1) and to the *Elevator Wrapper* (message

B1a in Figure 5.1) to reconfigure those component before the *Scheduler Wrapper* reconfigures the *Scheduler COTS Component*.

The reconfiguration plan generated by the *Scheduler Wrapper* for the Service Request function is depicted in Table 5.2.

Table 5.2: Reconfiguration Plan for Service Request Function in Scheduler Component

Reconfiguration Function	Reconfiguration Component	Reconfiguration Action
Service Request		
All Functions	Scheduler	Repair Block
Service Request		Repair Block
Service Request	Floor	Repair Block
Update Status Plan		Repair Block
Update Status Plan	Elevator	Repair Block
Schedule Request		Dependency Block
Elevator Button Request		Dependency Block
Update Elevator Status		Repair Block
Update Elevator Status	Elevator	Repair Block
Sensor Input		Dependency Block

The first line of the reconfiguration plan depicted in Table 5.2 states the name of the function that is malfunction. The following line establishes that all operations have to be ‘repair block’ in the *Scheduler COTS Component*. (The status of these operations is maintained in the Reconfiguration Manager described in chapter 7). The third, fifth, seventh, eighth, ninth, and eleventh lines that do not have a reconfiguration component, are used for the *Scheduler Wrapper* to update the reconfiguration information of the functions provided by the *Scheduler COTS component* and its dependent functions (This operation status information is in the Modified Interface described in chapter 7). The forth, sixth, eighth, and twelfth lines that do have a reconfiguration component, are used for the *Scheduler Wrapper* to notify other components against the malfunction.

The reconfiguration of the *Floor COTS Component* and the *Elevator COTS Component* in response to the faulty *Scheduler COTS Component* is carried out by the *Floor Wrapper* and the *Elevator Wrapper* respectively. The *Floor Wrapper* specifies the functional dependency among functions in the Floor Reconfiguration Table (Table 5.3).

Table 5.3: Floor Reconfiguration Table

Function	Callee			Caller	
	Function	Function Block	Component	Function	Component
Floor Button Request	Service Request		Scheduler		
On Off Direction Lamp				Open Door	Elevator
				Close Door	Elevator
Off Floor Lamp				Open Door	Elevator

When the *Scheduler Wrapper* notifies to the *Floor Wrapper* that the **Service Request** function is malfunction (message B1 in Figure 5.1), the *Floor Wrapper* generates the reconfiguration plan for the *Floor COTS Component*. The *Floor Wrapper* has the **Service Request** function and the Scheduler component as a callee. The *Floor Wrapper* includes the corresponding callee function and component - **Service Request** function and Scheduler component - in the reconfiguration plan.

The reconfiguration plan generated by the *Floor Wrapper* for the Service Request function is depicted in Table 5.4.

Table 5.4: Reconfiguration Plan for Service Request Function in Floor Component

Reconfiguration Function	Reconfiguration Component	Reconfiguration Action
Service Request		
Service Request		Repair Block
Floor Button Request		Dependency Block
Floor Button Request	Floor	Dependency Block

The reconfiguration plan depicted in Table 5.4 specifies in its first line the name of the function that is malfunction. The second and third lines that do not have a reconfiguration component are used by the *Floor Wrapper* to update the reconfiguration information of those functions (in the Reconfiguration Manager in chapter 7). The fourth line specifies that the *Floor Wrapper* has to change the current status of the Floor Button Request function to ‘dependency block’ (in the Modified Interface described in chapter 7).

The *Elevator Wrapper* specifies the functional dependency among functions in the Elevator Reconfiguration Table (Table 5.5).

Table 5.5: Elevator Reconfiguration Table

Function	Callee			Caller	
	Function	Function Block	Component	Function	Component
Schedule Request	Update Status Plan		Scheduler	Service Request	Scheduler
Sensor Input	Update Elevator Status		Scheduler		
Run Motor					
Stop Motor					
Open Door		Off Floor Lamp	Floor		
		On Off Direction Lamp	Floor		
Close Door		On Off Direction Lamp	Floor		
Elevator Button Request	Update Status Plan		Scheduler		

When the *Scheduler Wrapper* notifies to the *Elevator Wrapper* that the **Update Status Plan** function is malfunction (message B1a in Figure 5.1), the *Elevator Wrapper* generates the reconfiguration plan for the *Elevator COTS Component*. The *Elevator Wrapper* has the **Update Status Plan** function and the Scheduler component as a callee. The *Elevator Wrapper* includes the corresponding callee function and component – **Update Status Plan** function and Scheduler component - in the reconfiguration plan.

The reconfiguration plan generated by the *Elevator Wrapper* for the Update Status Plan function is depicted in Table 5.6.

Table 5.6: Reconfiguration Plan for Update Status Plan Function in Elevator Component

Reconfiguration Function	Reconfiguration Component	Reconfiguration Action
Update Status Plan		
Update Status Plan		Repair Block
Schedule Request		Dependency Block
Schedule Request	Elevator	Dependency Block
Update Status Plan		Repair Block
Elevator Button Request		Dependency Block
Elevator Button Request	Elevator	Dependency Block

The reconfiguration plan depicted in Table 5.6 specifies in its first line the name of the function that is malfunction. The second, third and fifth lines that do not have a reconfiguration component are used by the *Elevator Wrapper* to update the reconfiguration information of those functions (in the Reconfiguration Manager in chapter 7). The forth and sixth lines specify that the *Elevator Wrapper* has to change the current status of the Schedule Request and Elevator Button Request functions to ‘dependency block’ (in the Modified Interface object described in chapter 7).

When the *Scheduler Wrapper* notifies to the *Elevator Wrapper* that the **Update Elevator Status** function is malfunction (message B1a in Figure 5.1), the *Elevator Wrapper* generates the reconfiguration plan for the *Elevator COTS Component*. The *Elevator Wrapper* has the **Update Elevator Status** function and the Scheduler component as a callee. The *Elevator Wrapper* includes the corresponding callee function and component – **Update Elevator Status** function and Scheduler component - in the reconfiguration plan.

The reconfiguration plan generated by the *Elevator Wrapper* for the Update Elevator Status function is depicted in Table 5.7.

Table 5.7: Reconfiguration Plan for Update Elevator Status Function in Elevator Component

Reconfiguration Function	Reconfiguration Component	Reconfiguration Action
Update Elevator Status		
Update Elevator Status		Repair Block
Sensor Input		Dependency Block
Sensor Input	Elevator	Dependency Block

The reconfiguration plan depicted in Table 5.7 specifies in its first line the name of the function that is malfunction. The second and third lines that do not have a reconfiguration component are used by the *Elevator Wrapper* to update the reconfiguration information of those functions (in the Reconfiguration Manager in chapter 7). The forth line specifies that the *Elevator Wrapper* has to change the current status of the Sensor Input function to ‘dependency block’ (in the Modified Interface object described in chapter 7).

5.2.2 Reconfiguration of Non-Critical Functions

When a non-critical function for the system is anomalous, the *COTS component wrapper* of the COTS component that provides that function generates a reconfiguration plan against the anomalous function. The COTS component wrapper needs to block only the faulty function as ‘malfunction block’. The dependent *COTS component wrappers* are notified from the COTS component having the faulty function. The *COTS component wrappers* of the dependent COTS components of the faulty function generate the reconfiguration plans against the anomalous function. Because the faulty function is non-critical for the system, the dependent COTS component wrapper blocks the dependent functions as ‘function block’. This special kind of blocking does not block the dependent function as ‘dependency block’ because the malfunction function is not critical to the system and the repair of the non-critical function is going to be done during a regular maintenance period. With this blocking, the dependent function can work normally and the ‘function block’ prevents the dependent function to call the non-critical malfunction function.

Figure 5.2 depicts the reconfiguration of a non-critical function of the distributed elevator system (Gomaa 2000) with multiple elevators using the collaboration diagram of UML (Booch, Rumbaugh, and Jacobson 2005; Rumbaugh, Booch, and Jacobson 2005).

Figure 5.2: Reconfiguration for a Non-Critical Function of the Elevator System

When the *Floor Wrapper* detects a malfunction in the **Off Direction Lamp** function (chapter 4), the *Floor Wrapper* generates the corresponding reconfiguration plan for the non-critical **Off Direction Lamp** function. Because the **Off Direction Lamp** function is a non-critical function for the Elevator System, the repair of the *Floor COTS Component* could be done during the regular maintenance period. The *Floor COTS Component* may provide a degraded level of service until the maintenance time. To generate the reconfiguration plan, the *Floor Wrapper* contains the functional dependencies between each function in the *Floor COTS Component* and the functions in different COTS components. In these functional dependencies, the *Elevator COTS Component* has functions that are dependent of the functions provided by the *Floor COTS Component*. The *Floor COTS Component* must notify the dependent COTS components so that the COTS components do not request that anomalous function. The *Floor Wrapper* sends a notification to the *Elevator Wrapper* (message C1) saying that the **Off Floor Lamp** function is not working properly. When the *Floor Wrapper* receives the notification from the *Elevator Wrapper* (message C2) indicating that the COTS

component has been reconfigured, the *Floor Wrapper* proceeds to reconfigure the *Floor COTS Component* blocking the **Off Floor Lamp** function as ‘malfunction block’.

The *Floor Wrapper* maintains the functional dependency among functions in the *Floor COTS Component*, *Scheduler COTS Component*, and *Elevator COTS Component* in the Floor Reconfiguration Table (Table 5.2). For each function provided by the *Floor COTS Component*, the Floor Reconfiguration Table specifies the callee and caller functions and components. To generate the reconfiguration plan for the **Off Floor Lamp** function, the *Floor Wrapper* determines that the **Off Floor Lamp** function is non-critical for the system. The remaining procedure for generating reconfiguration plans in the Floor COTS component and other COTS components are the same as the malfunction of critical function.

Table 5.8 shows the reconfiguration plan for the faulty **Off Floor Lamp** function generated by the *Floor Wrapper*; whereas Table 5.9 depicts the reconfiguration plan for the same **Off Floor Lamp** function generated by the *Elevator Wrapper*.

Table 5.8: Reconfiguration Plan for Off Floor Lamp Function in Floor Component

Reconfiguration Function	Reconfiguration Component	Reconfiguration Action
Off Floor Lamp		
Off Floor Lamp		Malfunction Block
Off Floor Lamp	Floor	Malfunction Block
Off Floor Lamp	Elevator	Malfunction Block
Open Door		Dependency Block

The reconfiguration plan depicted in Table 5.8 specifies in its first line the name of the function that is malfunction. The second and fifth lines that do not have a reconfiguration component are used by the *Floor Wrapper* to update the reconfiguration

information of those functions. The third line specifies that the *Floor Wrapper* has to change the current status of the Off Floor Lamp function to ‘malfunction block’ (in the Modified Interface object described in chapter 7). The forth line is used for the *Floor Wrapper* to communicate to the *Elevator Component* the malfunction function.

Table 5.9: Reconfiguration Plan for Off Floor Lamp Function in Elevator Component

Reconfiguration Function	Reconfiguration Component	Reconfiguration Action
Off Floor Lamp		
Off Floor Lamp		Malfunction Block
Off Floor Lamp	Elevator	Function Block

The reconfiguration plan depicted in Table 5.9 specifies in its first line the name of the function that is malfunction. The second line that does not have a reconfiguration component is used by the *Elevator Wrapper* to update the reconfiguration information of this function. The third line specifies that the *Elevator Wrapper* has to change the current status of Off Floor Lamp function to ‘function block’.

5.3 High-Level View of Algorithm for Reconfiguration Mechanism

Based on the overview presented in the previous section, a high-level view of the reconfiguration mechanism can be described as follows:

- 1: Once the *COTS component wrapper* detects a malfunction in one of the functions provided by a COTS component (the detection mechanism has been described in chapter 4), the *COTS component wrapper* determines the criticality of the function.

- 2: If the malfunction function is critical for the system, the COTS component has to be repair immediately. The *COTS component wrapper* generates a reconfiguration plan for ‘repair block’. If the faulty function is non-critical for the system, the *COTS component wrapper* generates a reconfiguration plan for ‘malfunction block’.
- 3: The *COTS component wrapper* generates the reconfiguration plan using the Reconfiguration Table. The first action to be executed is to block the function (‘malfunction block’) or functions (‘repair block’). If the function has caller functions and components, the caller component should be notified for dependency block or function block in response to the anomaly function.
- 4: The *COTS component wrapper* executes the reconfiguration plan, blocking the function in the COTS component and notifying the other *COTS component wrappers*, so that they reconfigure the COTS component for ‘dependency block’.
- 5: The dependent component wrappers generate their reconfiguration plan as follows. If the faulty function and COTS component are found in callee, then the reconfiguration plans includes the function and the action for ‘dependency block’ or ‘function block’.
- 6: The other *COTS component wrappers* execute the reconfiguration plans and notify the faulty *COTS component wrapper* when the COTS components have been reconfigured.
- 7: When the faulty *COTS component wrapper* receives the notification from the other *COTS component wrappers*, the faulty *COTS component wrapper*

determines that the reconfiguration process for the whole system has been performed and the repair mechanism of the anomalous COTS component can start. The repair mechanism is described in chapter 6.

5.4 Summary

This chapter describes the reconfiguration mechanism for self-managed COTS components-based systems.

CHAPTER 6

REPAIR MECHANISM

6.1 Introduction

This chapter develops a mechanism that is capable of fixing anomalies in the COTS components. The *COTS component wrapper* repairs the COTS component containing anomalies. Some anomalies in a COTS component are critical to the whole system, which may lead to system failures. The repair of anomalies in COTS component is to protect the system from system failures. The *COTS component wrapper* heals the anomalous COTS component into a healthy COTS component. An anomalous function of a COTS component is repaired by the *COTS component wrapper* by means of re-initialization, re-installation, or modification of input to the component. The anomalies in a COTS component are repaired at the level of a component, not at the level of each function of a component. This is because a COTS component is a black box whose internal details are unknown enough to heal each function provided by a COTS component.

This chapter starts with describing the repair techniques in Section 6.2. Section 6.3 describes the repair mechanism. Section 6.4 provides a high-level view of the algorithm for the repair mechanism. Section 6.5 summarizes this chapter.

6.2 Repair Techniques

An anomalous COTS component is repaired by the *COTS component wrapper* by means of re-initialization, re-installation, or modification of input to the COTS component. An anomalous COTS component can be re-initialized if the COTS component provides a function for initializing the component state. An anomalous COTS component can also be re-installed using the same version of the COTS component or replaced with a variant of the COTS component if it is available. The variants of a COTS component can be different versions of the COTS component, which may either be implemented differently or provide different performances. Some inputs to a COTS component may need to be modified to fix anomalous functions in the COTS component.

The *COTS component wrapper* contains the repair techniques that are available for repairing the anomalous COTS component. The repair techniques can be one of the techniques described in this section or any other technique that is available. Table 6.1 shows the Floor Repair Techniques used by the *Floor component wrapper* in the Distributed Elevator System (Gomaa 2000) with multiple elevators.

Table 6.1: Floor Repair Techniques

Technique	Status
Re-install component	Non-used
Replace component	Non-used

The techniques available to repair a COTS component are executed sequentially, so that the first technique in the table (Table 6.1) must be performed for repairing the Floor COTS component. The *Floor component wrapper* also keeps track of which repair

technique has been used, so that the *Floor component wrapper* does not use the same technique again later.

The *Floor component wrapper* starts with repairing the component using the first technique from the Table 6.1 that has not been used yet. If the technique selected is ‘Re-install component’, the *Floor component wrapper* re-installs the same component using a copy of the Floor COTS component kept in a safe place and creates a new Floor COTS component. If the technique selected is the replacement of the component with a variant, ‘Replace component’, the *Floor component wrapper* selects a variant version of the COTS component as a new Floor COTS component.

6.3 Repair Mechanism

A repaired COTS component may lose the component state or data encapsulated in the COTS component before it is repaired. A state dependent COTS component maintains the state information of component, which may be changed as a message (or event) arrives at the COTS component. An example of state dependent COTS components is the Elevator COTS component in the Distributed Elevator System (Gomaa 2000), which loses its current state of the elevator after its repair. Some COTS components maintain data, which are used to produce a service in response to a request. Those data in a COTS component are lost after repair of the component.

A COTS component that is state-dependent or encapsulates data takes a time to reach at a stabilized state after it is repaired. During the stabilization of a repaired COTS component, the COTS component may provide low quality of services. The Scheduler COTS component in the Distributed Elevator System (Gomaa 2000) with multiple

elevators selects an elevator using data associated with each elevator, such as its current location and floors to be visited. When the Scheduler COTS component is repaired, it loses its current data and selects an elevator at random to send it to a floor in response to an elevator request. The Distributed Elevator System continues to provide acceptable services although the services are not high quality. However, the repaired Scheduler COTS component receives information about elevators from each elevator gradually, and it becomes stable enough to provide high quality of services.

When critical functions in the COTS component come to be anomalous, the COTS components should be repaired immediately. Each function in a COTS component can be classified into either a non-critical or critical function. For example, in the Distributed Elevator System (Gomaa 2000) with multiple elevators, a function that controls the elevator direction lamp at a floor can be a non-critical function, whereas a function scheduling multiple elevators can be a critical function. Non-critical functions in a COTS component may not result in system failures although the COTS components containing anomalous non-critical function may provide degraded services. Non-critical functions may be fixed at the regular maintenance period instead of immediate repair of the COTS component. This action is taken to reduce the risk of inconsistency of COTS component state that may result from the repair of the function. Furthermore, the regular maintenance can avoid the interruption of services provided by a COTS component that are not affected from the anomalous, non-critical functions. However, critical functions should be healed immediately although the COTS component repair may result in the inconsistency of COTS component states between before and after repairing the COTS component. System failures are more critical than the inconsistency of a component state.

A COTS component that has anomalous functions is isolated from the neighboring components before its repair, so that the repair is not interrupted by other COTS component requests. The isolation of the anomalous COTS component (described in chapter 5) is realized by blocking all functions in the anomalous COTS component, and blocking all functions in the neighboring COTS components that require services from the component being repaired (Gomaa 2000; Kramer and Magee 1990). Before blocking functions, the COTS components should wait until the functions that are currently handling service requests are finished completely. Otherwise, if the functions are interrupted during their executions, the system fails into an inconsistent state.

6.4 High-Level View of Algorithm for Repair Mechanism

Based on the overview presented in the previous section, a high-level view of the algorithm for the repair mechanism can be described as follows:

- 1: Once the malfunction COTS component and their dependent COTS components have been reconfigured for repair (the reconfiguration mechanism has been described in chapter 5), the malfunction *COTS component wrapper* starts the repair process.
- 2: The *COTS component wrapper* selects the first repair technique that has not been used yet using the Repair Techniques Table (Table 6.1).
- 3: The *COTS component wrapper* executes the technique selected in the previous step.
- 4: The *COTS component wrapper* updates the status of the technique performed as ‘used’ in the Repair Techniques Table (Table 6.1).

- 5: As the *COTS component wrapper* repairs the faulty COTS component, the repaired COTS component and dependent COTS components have to be reconfigured again. They ‘unblock’ the functions that were blocked during the reconfiguration before repair. The description of this process can be found in chapter 5.

6.5 Summary

This chapter has developed the repair mechanism for self-managed COTS components-based systems. An anomalous COTS component can be repaired by the *COTS component wrapper* by re-initialization, re-installation, or replacement of the COTS component, as well as using modification of inputs to the COTS component.

CHAPTER 7

DESIGN OF A COTS COMPONENT WRAPPER FOR SELF-MANAGING COTS COMPONENTS

7.1 Introduction

This chapter presents the design of the *COTS component wrapper* for self-managing COTS components. A self-managed COTS component is designed using a wrapper (DeLine 1999), which encapsulates the COTS component to control the interaction between COTS components. In the design of a *COTS component wrapper* for self-managing COTS component, the following requirements are needed:

- Detection: Each *COTS component wrapper* needs to be capable of monitoring a COTS component to detect anomalies that may result in system failures.
- Reconfiguration: COTS components need to be reconfigured at run-time by their *COTS component wrappers* to minimize the impact from the anomalies detected. As the anomalous COTS component is back to normal after repair, *COTS component wrappers* also need to reconfigure the system to provide full functionality of the system.
- Repair: Each *COTS component wrappers* needs to repair anomalies in the COTS component being monitored.

In addition to the requirements presented above, each COTS component needs to be used as sold by COTS component vendors. That is, a COTS component is considered

a black box whose internal details are hidden except for its original interface to services the component provides and requires.

A *COTS component wrapper* of self-managing COTS components is structured into several objects to support the detection, reconfiguration and repair mechanisms: COTS Modified Interface, COTS Monitor, Wrapper Controller, Reconfiguration Manager, and Repair Manager.

By consideration of reusability, application-independent algorithms in the objects of a *COTS component wrapper* are implemented separately from application-dependent information. Application-independent algorithms of a *COTS component wrapper* are common to all COTS components so that they are reused for different COTS components. Application-dependent part of a *COTS component wrapper* contains information specific to a COTS component, which should be updated for a different COTS component.

This chapter starts by describing the requirements for designing a COTS component wrapper in Section 7.2. Section 7.3 presents the design of COTS component wrappers. Section 7.4 presents the interaction of the objects in the COTS component wrappers. Section 7.5 summarizes this chapter.

7.2 Requirements for Designing a COTS Component Wrapper

COTS component wrappers for COTS components play an important role for self-management of COTS component-based systems. *COTS component wrapper* are involved in detection of anomalies in the component (described in chapter 4), dynamic reconfiguration of COTS component systems (described in chapter 5), and repair of

anomalous COTS component (described in chapter 6), so that COTS component-based systems remain more dependable.

A *COTS component wrapper* needs to be designed in order to support the properties of self-management –detection, reconfiguration, and repair. Therefore, a COTS component wrapper necessitates to provide specific functionalities to support each one of the mechanisms: detection, reconfiguration, and repair.

7.2.1 COTS Component Wrapper Functionalities to Support the Detection

Mechanism

The *COTS component wrapper* shall provide the following functionalities in order to monitor and detect anomalies in a COTS component:

- a) The *COTS component wrapper* shall identify a function request with its corresponding parameters from other COTS component, or an external user such as a human actor, or external device.
- b) The *COTS component wrapper* shall monitor each function provided by the COTS component being wrapped.
- c) The *COTS component wrapper* shall call a function from the COTS component being wrapped only when the function performs its activity normally.
- d) The *COTS component wrapper* shall capture the responses and its arguments from the COTS component being wrapped.
- e) The *COTS component wrapper* shall identify the corresponding successor function for each function provided by the COTS component being wrapped.

- f) The *COTS component wrapper* shall detect anomalies in the COTS component being wrapped.

7.2.2 COTS Component Wrapper Functionalities to Support the Reconfiguration Mechanism

The *COTS component wrapper* shall provide the following functionalities in order to reconfigure the COTS component wrapped and the other components in the system:

- a) The *COTS component wrapper* shall identify whether the anomalous function is critical or non-critical for the system, and shall make a decision about ‘malfunction block’ or ‘repair block’ against the anomalous function based on its criticality.
- b) The *COTS component wrapper* shall generate a reconfiguration plan against the anomalous function and for the action to be performed (‘malfunction block’ or ‘repair block’) to reconfigure not only the component whose function is malfunction, but also the functions provided by other components that are dependent of the anomalous function.
- c) The *COTS component wrapper* shall receive information about the anomalous function from other COTS components in the system and shall generate the corresponding reconfiguration plan.
- d) The *COTS component wrapper* shall execute the reconfiguration plan so that it blocks or unblocks functions, updating the reconfiguration information of the functions specified in the reconfiguration plan, and notifying the

anomalous function to the dependent COTS components specified in the reconfiguration plan.

- e) The *COTS component wrapper* shall generate and execute a reconfiguration plan after the COTS component has been repaired. This plan is needed to unblock all the functions that were blocked during the previous reconfiguration process, not only in the COTS component wrapped but also in the dependent COTS components.

7.2.3 COTS Component Wrapper Functionalities to Support the Repair Mechanism

The *COTS component wrapper* shall provide the following functionalities in order to repair the COTS component wrapped:

- a) The *COTS component wrapper* shall select the first repair technique that has not been used yet in order to repair the COTS component.
- b) The *COTS component wrapper* shall execute the repair technique that has been selected to create a new COTS component, marking this repair technique as used.
- c) The *COTS component wrapper* shall use the new COTS component previously created.
- d) The *COTS component wrapper* shall start a new reconfiguration process to unblock the functions that were blocked because of the repairing.

7.3 Design of COTS Component Wrappers

A COTS component wrapper of self-managing COTS component is structured into several objects to support the properties of self-management of COTS components. Figure 7.1 depicts the COTS component wrapper architecture for self-management of COTS components. The objects constituting the COTS component wrapper for COTS components co-operate with each other to provide the functionalities described in section 7.2. The details of each object are given in the following sections.

Figure 7.1: COTS Component Wrapper Architecture for Self-Managed COTS Component

7.3.1 COTS Modified Interface

The COTS Modified Interface is a linker between a *COTS component wrapper* and its COTS component. The interface to an original COTS component is wrapped up by the COTS Modified Interface through which the COTS component provides services

to and/or requires them from other COTS components. The COTS Modified Interface captures the messages passing between the COTS component and other COTS components. The messages are delivered to the COTS Monitor, which detects anomalies in the COTS component using the messages.

The COTS Modified Interface maintains the status of each function provided by a COTS component. A function is ‘unblocked’ if it performs its functionality normally, whereas a function is marked ‘blocked’ if either the function is anomalous (referred to as malfunction block) or the function requires anomalous function in other COTS component (referred to as dependency block). A function can be blocked during the repair of anomalous COTS component as well (referred to as repair block). When a COTS component (service requestor) requests a function from other COTS component (service provider), the COTS Modified Interface for the service provider COTS component checks the status of the function requested, and then it allows the function to be performed if the function is unblocked.

The COTS Modified Interface is structured into one active object, the COTS Modified Interface object, and five passive objects: Operation Request Parameter, Operation Status, Function Status, Operation Return Type, and Successor Operation objects.

Operation Request Parameter: The Operation Request Parameter entity contains the parameters information for each function provided by the COTS component wrapped. The COTS Modified Interface can identify the function call’s parameters, and be capable of not only showing the information to the console but also calling the

function with its arguments from the COTS component. Figure 7.2 depicts the attributes of the Operation Request Parameter entity class.

Figure 7.2: Operation Request Parameter Entity Class

Operation Status: The Operation Status entity contains the status of each function provided by the COTS component wrapped. Figure 7.3 shows the attributes of the Operation Status entity class.

Figure 7.3: Operation Status Entity Class

Function Status: The Function Status entity contains the status of the successor function of each function provided by the COTS component wrapped. The COTS Modified Interface will call the successor function only if its status is ‘unblock’. Figure 7.4 illustrates the attributes of the Function Status entity class.

Figure 7.4: Function Status Entity Class

Operation Return Type: The Operation Return Type entity contains the parameter information of the return of each function provided by the COTS component wrapped. The COTS Modified Interface can identify the parameter information returned by the COTS component wrapped and be able not only to show the information to the console but also to call the successor function with its arguments. Figure 7.5 depicts the attributes of the Operation Return Type entity class.

<<entity>> Operation Return Type
operationName: String parameterName: String parameterType: String parameterOrder: Integer

Figure 7.5: Operation Return Type Entity Class

Successor Operation: The Successor Operation entity contains the successor function information of each function provided by the COTS component wrapped. This entity class shows a successor operation and its component if an operation has a successor operation. Figure 7.6 shows the attributes of the Successor Operation entity class.

<<entity>> Successor Operation
operationName: String successorOperationName: String successorOperationComponent: String

Figure 7.6: Successor Operation Entity Class

7.3.2 COTS Monitor

The functions provided by a COTS component are monitored by the COTS Monitor, which encapsulates statecharts describing the specifications of each function of the COTS components. When a function in a service provider COTS component is invoked by a service requestor COTS component, the statechart for the function makes a transition from the 'Idle' state to the next state. Some function in a COTS component may need services from other COTS components to complete its own service. The statechart for a function also describes these dependencies among the functions in COTS components.

The statecharts for each function of a COTS component, encapsulated in the COTS Monitor, are executed by the messages from the COTS Modified Interface. The interface to a COTS component is invoked by the COTS Modified Interface, which is invoked by service requestor COTS component. The COTS Modified Interface is an intermediary in interaction between a COTS component and its service requestor/provider COTS component, notifying the COTS Monitor when the COTS components communicate with each other. The COTS Monitor uses the notification messages to check the specifications of functions so that it determines when and which function in a COTS component is anomalous. The COTS Monitor presumes that a function provided by a COTS component is anomalous if the expected notification messages from the COTS Modified Interface have not arrived within a reasonable time interval. Each state in a statechart for a function, encapsulated in the COTS Monitor, should make transitions within an expected time interval once the function is invoked.

Otherwise, the COTS Monitor reports the anomaly of the function to the Wrapper Controller, which in turns takes an appropriate action against the anomalous function.

The COTS Monitor is structured into two active objects - the COTS Monitor and a COTS Monitor Timer objects - and two passive objects - Operation Statechart and Operation Statechart Information objects.

Operation Statechart: The Operation Statechart entity contains the information needed to execute the statecharts of each operation provided by the COTS component wrapped. Figure 7.7 shows the attributes of the Operation Statechart entity class.

Figure 7.7: Operation Statechart Entity Class

Operation Statechart Information: The Operation Statechart Information entity contains the times needed to monitor the transition of state of the statechart of each operation provided by the COTS component wrapped. Figure 7.8 depicts the attributes of the Operation Statechart Information entity class.

Figure 7.8: Operation Statechart Information Entity Class

7.3.3 Reconfiguration Manager

The reconfiguration of a self-managed COTS component is performed by the Reconfiguration Manager in a *COTS component wrapper*. The COTS components are reconfigured by blocking or unblocking the interfaces to each function of COTS components, which are encapsulated in the COTS Modified Interface. Anomalous functions are blocked so that other COTS components cannot request the functions paralyzed. A function is blocked either if the function is anomalous, or if the function requires anomalous functions in different COTS components. The status of a function is changed to ‘unblocked’ as the function resumes its service normally.

The Reconfiguration Manager has information associated with the reconfiguration of a COTS component. The information includes the status of both each function in a COTS component and its associated functions (i.e., callee and caller functions) in other COTS components. A function may call some functions (i.e., callee functions) in other COTS components, or may be called by some functions (i.e., caller functions) in other

COTS components. The status of each function (including its callee and caller functions) is classified into ‘unblock’, ‘malfunction block’, ‘dependency block’, ‘function block’, and ‘repair block’. In addition, the Reconfiguration Manager contains the dependency relationship among functions in a COTS component and in other COTS components. This relationship is used to generate reconfiguration plans against anomalous functions both within a COTS component and in different COTS components.

The COTS components are reconfigured into the original configuration again when anomalous COTS components resume their services. After repairing the anomalous functions, the functions that have been blocked come to be unblocked to provide services, and the COTS components dependent on the blocked COTS component re-start communicating with the COTS component resumed.

The Reconfiguration Manager is structured into two active objects - the Reconfiguration Plan Generator and Reconfiguration Plan Executor objects - and three passive objects - Operation Status, Operation Caller and Operation Callee objects.

Operation Reconfiguration Status: The Operation Reconfiguration Status entity class contains the status of functions in the COTS component wrapped and its associated functions (i.e., callee and caller functions) in other COTS components. Figure 7.9 illustrates the attributes of the Operation Reconfiguration Status entity class.

Figure 7.9: Operation Reconfiguration Status Entity Class

Operation Caller: The Operation Caller entity class contains the dependency relationship (who is the caller function and what COTS component provides it) among functions in the COTS component wrapped and in other COTS components. Figure 7.10 shows the attributes of the Operation Caller entity class.

Figure 7.10: Operation Caller Entity Class

Operation Callee: The Operation Callee entity class contains the dependency relationship (who is the callee function and what COTS component provides it) among functions in the COTS component wrapped and in other COTS components. Figure 7.11 depicts the attributes of the Operation Callee entity class.

Figure 7.11: Operation Callee Entity Class

7.3.4 Repair Manager

An anomalous function of a COTS component is repaired by the Repair Manager by means of re-initialization, re-installation, or modification of input to the COTS component. The anomalies in a COTS component are repaired at the level of a component, not at the level of each function of a component. This is because a COTS component is a black box whose internal details are unknown enough to heal each function provided by the COTS component.

The Repair Manager contains information related to the different techniques that are available for repairing the anomalous COTS component. These techniques are executed sequentially, so the first technique that has not been used yet must be performed first for repairing the COTS component.

The Repair Manager is structured into one active object, the Repair Manager object, and one passive object, the Repair Technique object.

Repair Technique: The Repair Technique entity class contains information related to the repair techniques available to repair the COTS component wrapped. Figure 7.12 shows the attributes of the Repair Technique entity class.

Figure 7.12: Repair Technique Entity Class

7.3.5 Wrapper Controller

The Wrapper Controller coordinates the other objects in a *COTS component wrapper* to control the self-management of a COTS component. The Wrapper Controllers in *COTS component wrappers* also co-operate with each other to reconfigure the system at run-time against anomalies detected. The Reconfiguration Managers in *COTS component wrappers* co-operate with each other via the Wrapper Controller to reconfigure COTS components, so that the neighboring COTS components have the minimal impact from anomalous COTS components.

The Wrapper Controller in a wrapper for a COTS component determines whether a function in a COTS component should be repaired immediately or not. The Wrapper Controller maintains information on function type such as critical or non-critical functions. With the notification messages about anomalous functions from the COTS Monitor, the Wrapper Controller makes decisions about what actions will be taken against the functions – either just blocking anomalous functions, or both blocking and repairing the whole COTS component immediately.

The Wrapper Controller is structured into one active object, the Wrapper Controller object, and one passive object, the Operation Criticality object.

Operation Criticality: The Operation Criticality entity class contains the criticality of each operation provided by the COTS component wrapped. Figure 7.13 depicts the attributes of the Operation Criticality entity class.

Figure 7.13: Operation Criticality Entity Class

7.4 Interaction of the Objects in the COTS Component Wrapper

The objects in the *COTS component wrapper* interact with each other to monitor and detect anomalies in a COTS component, dynamically reconfigure the anomalous COTS component and the other COTS components in the system, and repair the anomalous COTS component. The following sections present the interaction of the objects in the *COTS component wrapper* which is described by mechanism: detection, reconfiguration and repair respectively.

7.4.1 Detection Mechanism

The *COTS component wrapper* performs the activities described in the Detection Mechanism (chapter 4) using the following objects: The COTS Modified Interface and the COTS Monitor. The relationships between the COTS Modified Interface and the COTS Monitor as well as their passive objects are depicted using the UML class diagram (Booch, Rumbaugh, and Jacobson 2005; Rumbaugh, Booch, and Jacobson 2005). A COTS component interacts with its *COTS component wrapper*, and a service requestor COTS component, the Primary Actor, who requests an operation from the COTS

component via the COTS Modified Interface, as shown in Figure 7.14. The COTS Modified Interface reads the parameters information of the function requested from the Operation Request Parameter entity, checks the status of the function requested from the Operation Status entity, and notifies the events happening in the COTS Modified Interface to the COTS Monitor. The COTS Monitor maintains the statecharts of each function in the Operation Statechart entity, and updates the information in the Operation Statechart Information entity. The COTS Modified Interface reads the parameters from the Operation Return Type entity and the successor function information from the Successor Operation entity, and checks the status of the successor function in the Function Operation entity.

Figure 7.14: Detection Mechanism Class Diagram

The UML communication diagram (Booch, Rumbaugh, and Jacobson 2005; Rumbaugh, Booch, and Jacobson 2005) is developed to show how objects collaborate with each other to support the detection mechanism. Figure 7.15 depicts the Detection Mechanism Communication Diagram.

Figure 7.15: Detection Mechanism Communication Diagram

The message sequence description corresponding to the communication diagram depicted in Figure 7.15 is as follows:

D1: The **Operation Request** arrives at the *COTS Modified Interface* object.

D2: The *COTS Modified Interface* extracts the function name requested, and **Reads** its corresponding **Parameters** from the *Operation Request Parameter* object to identity the name and type of the parameters. The *COTS Modified*

Interface can show those parameters to the console and call the function from the COTS component.

D3: The *COTS Modified Interface* **Checks** the **Status** of the function requested.

D4: If the status of the function requested is 'unblock', the *COTS Modified Interface* sends the **Operation Arrived** message to the *COTS Monitor*. If the status of the function is different from 'unblock', the function is not allowed into the wrapper.

D4.1: The *COTS Monitor* **Reads** the current **State** and next state for the function arrived from the *Operation Statechart* object.

D4.2: The *COTS Monitor* **Updates** the state information on the *Operation Statechart Information* object.

D5: The *COTS Modified Interface* **Requests** the corresponding **Operation** from the *COTS component*.

D6: The *COTS Modified Interface* sends the **Operation Requested** message to the *COTS Monitor*.

D6.1: The *COTS Monitor* **Reads** the current **State** and next state for the function requested from the *Operation Statechart* object.

D6.2: The *COTS Monitor* **Updates** the state information on the *Operation Statechart Information* object, and starts timing the execution of the requested function.

D7: The *COTS Modified Interface* receives the **Response** from the *COTS component*.

D7a: The *COTS Monitor* waits for a specific time interval for the ‘response arrived’ message from the *COTS Modified Interface*. If this time expires, the *COTS Monitor* detects that the function requested is malfunction, and sends a **Notify Failure** message to the *Reconfiguration Mechanism*.

D8: The *COTS Modified Interface* **Reads** the *COTS component* response **Parameters** from the *Operation Return Type* object to identity the name and type of the parameters of successor function called by the COTS component.

D9: The *COTS Modified Interface* sends the **Response Arrived** message to the *COTS Monitor*.

D9.1: The *COTS Monitor* **Reads** the current **State** and next state for the response requested from the *Operation Statechart* object.

D9.2: The *COTS Monitor* **Updates** the state information on the *Operation Statechart Information* object.

D10: The *COTS Modified Interface* **Reads** the **Next Operation** from the *Successor Operation* object.

D11: If a successor function exists, the *COTS Modified Interface* **checks** the **Status** of the successor function.

D12: If a successor function exists and its status is ‘unblock’, the *COTS Modified Interface* **Requests** the **Next Operation** from the corresponding COTS component. If there is no successor function, the process stops at this sequence.

D13: The *COTS Modified Interface* sends the **Next Operation Called** message to the *COTS Monitor*.

D13.1: The *COTS Monitor* **Reads** the current **State** and next state for the next function called from the *Operation Statechart* object.

D13.2: The *COTS Monitor* **Updates** the state information on the *Operation Statechart Information* object.

7.4.2 Reconfiguration Mechanism

The *COTS component wrapper* performs the activities described in the Reconfiguration Mechanism (chapter 5) using the following objects: the Wrapper Controller, the Reconfiguration Manager, the COTS Modified Interface, and the COTS Monitor. The relationships between the Wrapper Controller, the Reconfiguration Manager, the COTS Modified Interface, and the COTS Monitor as well as their passive objects are depicted using the UML class diagram (Booch, Rumbaugh, and Jacobson 2005; Rumbaugh, Booch, and Jacobson 2005). A *COTS component wrapper* interacts with other *COTS component wrappers* to reconfigure COTS components at run-time, as shown in Figure 7.16. The COTS Monitor notifies the Wrapper Controller when a function in a COTS component is malfunction. With this notification, the Wrapper Controller checks the criticality of the function to determine the action to be taken – malfunction block or repair block. The Wrapper Controller interacts with the Reconfiguration Manager so that it reconfigures the system against the anomalous function. The Reconfiguration Manager generates the reconfiguration plan based on the information read from the Operation Caller and Operation Callee entities. The Reconfiguration Manager executes the reconfiguration plan and interacts with the COTS Modified Interface that updates the function's status with the new status provided by the

reconfiguration plan. When the reconfiguration plan is executed after repair, the Reconfiguration Manager also interacts with the COTS Monitor which updates the Operation Statechart Information entity.

Figure 7.16: Reconfiguration Mechanism Class Diagram

The UML communication diagram (Booch, Rumbaugh, and Jacobson 2005; Rumbaugh, Booch, and Jacobson 2005) is developed to show how objects collaborate with each other to support the reconfiguration mechanism. Figure 7.17 depicts the communication diagram for the Reconfiguration Mechanism.

Figure 7.17: Reconfiguration Mechanism Communication Diagram

The message sequence description corresponding to the communication diagram depicted in Figure 7.17 is as follows:

- R1: The *COTS Monitor* has detected a failure in a function (message D7a in Figure 7.15) and sends a **Notify Failure** message to the *Wrapper Controller*.
- R2: The *Wrapper Controller* **Reads** the **Criticality** of the anomalous function from the *Operation Criticality* object.

R3: The *Wrapper Controller* **Asks a Reconfiguration Plan** for the anomalous function to the *Reconfiguration Manager*.

R4: If the anomalous function is provided by the COTS component wrapped, the *Reconfiguration Manager* **Reads the Caller Information** from the *Operation Caller* object to determine whether there are caller functions and their COTS components. The caller functions should be blocked as ‘dependency block’ in response to the anomalous function.

R5: If the anomalous function is provided by other COTS components, the *Reconfiguration Manager* **Reads the Callee Information** from the *Operation Callee* object to identify what function in the COTS component wrapped has to be blocked as ‘dependency block’.

R6: The *Reconfiguration Manager* generates the reconfiguration plan and starts executing the plan, **updating the Status** of the functions in the *Operation Reconfiguration Status* object.

R7: If the reconfiguration plan states that other COTS components need to be reconfigured, the *Reconfiguration Manager* sends a **Reconfigure Other Components** message to the *Wrapper Controller*.

R8: The *Wrapper Controller* sends a **Reconfigure Other Components** messages to the neighboring COTS components.

R9: The *Reconfiguration Manager* sends a message to the *COTS Modified Interface* to **Block the Functions**.

R10: The *COTS Modified Interface* **Updates the Status** of the functions in the *Operation Status* object.

R11: The *COTS Modified Interface* sends a **Functions Blocked** message to the *Reconfiguration Manager*.

R12: The *Reconfiguration Manager* sends a **Plan Executed** message to the *Wrapper Controller*. If the anomalous function is provided by the COTS component wrapped and it is critical to the system, the *Wrapper Controller* needs to start the *Repair Mechanism*; otherwise, this is the end of the reconfiguration mechanism before the repair of the COTS component.

R13: After the *Repair Mechanism* has been executed, the *Wrapper Controller* starts the reconfiguration **Asking** to the *Reconfiguration Manager* for a **Reconfiguration Plan** to unblock all the functions that were blocked during the previous reconfiguration.

R14: If the function to be unblocked is provided by the COTS component wrapped, the *Reconfiguration Manager* **Reads** the **Caller Information** from the *Operation Caller* object to determine whether there are the caller functions and their COTS components. These caller functions are informed to be ‘unblock’.

R15: If the function to be unblocked is provided by other COTS components, the *Reconfiguration Manager* **Reads** the **Callee Information** from the *Operation Callee* object to identify what function in the COTS component wrapped has to be unblocked.

R16: The *Reconfiguration Manager* generates the reconfiguration plan and starts executing the plan, **updating** the **Status** of the functions in the *Operation Reconfiguration Status* object.

R17: If the reconfiguration plan states that other COTS components need to be reconfigured, the *Reconfiguration Manager* sends a **Reconfigure Other Components** message to the *Wrapper Controller*.

R18: The *Wrapper Controller* sends a **Reconfigure Other Components** messages to the neighboring COTS components.

R19: The *Reconfiguration Manager* sends a message to the *COTS Modified Interface* to **Unblock the Functions**.

R20: The *COTS Modified Interface* **Updates** the **Status** of the functions in the *Operation Status* object.

R21: The *COTS Modified Interface* sends a **Functions Unblocked** message to the *Reconfiguration Manager*.

R22: If the function to be unblocked is provided by the COTS component wrapped, the *Reconfiguration Manager* sends a **Reinitialize Statechart** message to the *COTS Monitor*.

R23: The *COTS Monitor* **Updates** the **Information** in the *Operation Statechart Information* object.

R24: The *COTS Monitor* sends a **Statechart Reinitialized** message to the *Reconfiguration Manager*.

R25: The *Reconfiguration Manager* sends a **Plan Executed** message to the *Wrapper Controller*.

7.4.3 Repair Mechanism

The *COTS component wrapper* performs the activities described in the Repair Mechanism (chapter 6) using the following objects: The Wrapper Controller, the Repair Manager, and the COTS Modified Interface. The relationships between the Wrapper Controller, the Repair Manager, the COTS Modified Interface, and their passive objects are depicted using the UML class diagram (Booch, Rumbaugh, and Jacobson 2005; Rumbaugh, Booch, and Jacobson 2005). The Wrapper Controller interacts with the Repair Manager to repair the COTS component, as shown in Figure 7.18. The Repair Manager selects the first repair technique that has not been used yet from the Repair Technique entity, and interacts with the COTS Modified Interface to send the COTS component repaired.

Figure 7.18: Repair Mechanism Class Diagram

The UML communication diagram (Booch, Rumbaugh, and Jacobson 2005; Rumbaugh, Booch, and Jacobson 2005) is developed to show how objects collaborate with each other to support the repair mechanism. Figure 7.19 depicts the Repair Mechanism Communication Diagram.

Figure 7.19: Repair Mechanism Communication Diagram

The message sequence description corresponding to the communication diagram depicted in Figure 7.19 is as follows:

P1: The *Wrapper Controller* sends a **Repair COTS** message to the *Repair Manager*.

P2: The *Repair Manager* **Selects** the first **Technique** that has not been used yet from the *Repair Technique* object.

P3: The *Repair Manager* executes the repair technique and creates a new COTS component. The *Repair Manager* sends the **New COTS** to the *COTS Modified Interface*.

P4: The *COTS Modified Interface* receives the new COTS component and sends an **Acknowledge** to the *Repair Manager*.

P5: The *Repair Manager* sends a **COTS Repaired** message to the *Wrapper Controller*. The *Wrapper Controller* starts the reconfiguration after repair process (message R13 in Figure 7.17).

7.6 Summary

This chapter has described the design of a *COTS component wrapper* for self-managing COTS components. A COTS component wrapper supporting self-management of each COTS component is structured to several objects –COTS Modified Interface, COTS Monitor, Wrapper Controller, Reconfiguration Manager, and Repair Manager. These objects are designed to realize the requirements of a *COTS component wrapper* for self-managing COTS components. Each COTS component wrapper is capable of detecting anomalies in the COTS component, reconfiguring the COTS component against the anomalies detected, and repairing the anomalous COTS component. The interaction among the objects in the *COTS component wrapper* to achieve the properties of self-

management of COTS component has been also described. By consideration of reusability, application-independent algorithms of the COTS component wrapper (see Appendix A) are implemented separately from application-dependent information. Application-independent algorithms implemented in objects constituting a COTS component wrapper are common to all COTS components, whereas application-dependent part is the information specific to each COTS component.

CHAPTER 8

DISTRIBUTED ELEVATOR SYSTEM: A CASE STUDY

8.1 Introduction

A distributed elevator system is one of the two case studies for validating the approach for self-managing COTS component-based systems. This case study first implements the *COTS component wrappers* and uses the COTS components developed by a Master Student of the Computer Science Department at Texas Tech University.

The software architecture of the distributed elevator system is structured into three Elevator COTS components, ten Floor COTS components, and one Scheduler COTS component. In the implementation of the *COTS component wrappers* corresponding to each one of the COTS components, the application-dependent information associated with each of the COTS components for the distributed elevator system is depicted by means of tables. To validate the self-managing mechanisms proposed in this approach, a series of test cases were used to test the system.

This chapter begins by describing the distributed elevator system used in this case study in Section 8.2. Section 8.3 shows the software architecture of the distributed elevator system. Section 8.4 presents the implementation of the self-managed distributed elevator system. Section 8.5 addresses the validation of this research. Section 8.6 summarizes this chapter.

8.2 Problem Description

To demonstrate self-managing of COTS component-based approach, a distributed elevator system is considered. The distributed elevator system with multiple elevators (Gomaa 2000) has to schedule elevators to respond to requests from users at multiple floors and control the motion of the elevators between floors. Therefore, the distributed elevator system is structured into three Elevator COTS components, ten Floor COTS components, and one Scheduler COTS component. For each Elevator COTS component, there are:

- A set of elevator buttons. A user presses an elevator button to select a destination.
- A corresponding set of elevator lamps. Elevator lamps indicate the floors to be visited by the elevator.
- An elevator motor. The elevator motor is controlled by commands to move up, move down, and stop.
- An elevator door. The elevator door is controlled by commands to open and close the door.
- An elevator sensor. The elevator sensor is a simulation of a real sensor indicating the movement of the elevator.

For each Floor COTS component, there are:

- Up and down floor buttons. A user presses a floor button to request an elevator.
- A corresponding pair of floor lamps. The floor lamps indicate the directions that the user has been requested.

At each floor, and for each elevator, there is a pair of direction lamps to indicate whether an arriving elevator is heading in the up or down direction. For top and bottom floors, there is only one floor button, one floor lamp, and (for each elevator) one direction lamp.

The Elevator COTS component is a control component, which provides overall coordination of devices such as elevator lamps, motor, door, sensor, and elevator buttons. The Floor COTS component collects floor button requests from elevator users and requests elevators to be sent to the floors from the Scheduler COTS component. The Scheduler COTS component schedules all elevators in a building in response to the elevator requests from the Floor COTS components.

8.3. Software Architecture

In the software architecture, each Elevator COTS component is allocated to an elevator, while each Floor COTS component is allocated to a floor. The Scheduler COTS component is allocated to a separate node for performance reasons, so that it can rapidly respond to elevator requests.

When an elevator user presses the floor button at a specific floor, the floor lamp is switched on by the hardware. The Floor COTS component switches off the floor lamp as an elevator arrives at the floor, indicating the elevator direction (up or down) by switching on the corresponding direction lamp at the floor. The Floor COTS Component is connected to three external devices: floor button, floor lamp, and direction lamp. As shown in Figure 8.1.

Figure 8.1: Floor COTS Component Software Architecture

The Floor COTS Component collects floor button requests from elevator users and requests elevators to be sent to the floors from the Scheduler COTS Component. The Scheduler COTS Component schedules all elevators in a building in response to the elevator user requests from the Floor COTS Component.

The Elevator COTS Component provides overall coordination of devices such as motor, door, and elevator buttons as shown in Figure 8.2. When the Scheduler COTS Component schedules a floor to be visited by the elevator, the Elevator COTS Component closes the door, and starts the motor up or down according to its position on the building. When the elevator arrives to the requested floor, the Elevator COTS Component stops the motor, opens the door, and communicates to the Floor COTS component its arrival at the floor, so the Floor COTS Component can power off the floor lamp and power on/off the direction lamp.

Figure 8.2: Elevator COTS Component Software Architecture

8.4 Implementation of the Distributed Elevator System

By consideration of reusability, the COTS component wrapper that plays a key role for self-managing COTS components is designed and implemented by means of both application-independent part and application-dependent part. Application independent part of a COTS component wrapper is common to all COTS components so that it is reused for different COTS component once it is implemented (see chapter 7). Application dependent part of a wrapper contains information specific to a COTS component, which should be updated for a different COTS component. Application-dependent COTS information is described in a COTS component wrapper by means of tables. The following sections presents the tables used by each one of the COTS component wrappers involved in this case study.

8.4.1 Floor COTS Component Wrapper

The Floor COTS component wrapper self-manages the Floor COTS component using the application independent algorithms presented in chapter 7 and the application-dependent information as described in the following sections. The application-dependent information has been implemented by means of tables and these tables have been assigned to the Floor COTS component wrapper's objects.

8.4.1.1 Floor Modified Interface Object

The Floor Modified Interface object provides the Floor Operation Status Table (Table 8.1), the Floor Operation Request Parameter Table (Table 8.2), the Floor Operation Return Type Table (Table 8.3), the Floor Successor Operation Table (Table 8.4), and the Floor Function Status Table (Table 8.5).

Table 8.1: Floor Operation Status Table

Operation	Status
floorButtonRequest	Unblock
offFloorLamp	Unblock
onOffDirectionLamp	Unblock

Table 8.2: Floor Operation Request Parameter Table

Operation	Parameter	Type	Order
floorButtonRequest	Floor No.	integer	1
floorButtonRequest	Direction	string	2
offFloorLamp	Floor No.	integer	1
offFloorLamp	Direction	string	2
offFloorLamp	Elevator No.	integer	3
onOffDirectionLamp	Floor No.	integer	1
onOffDirectionLamp	Status	string	2
onOffDirectionLamp	Elevator No.	integer	3

Table 8.3: Floor Operation Return Type Table

Operation	Parameter	Type	Order
floorButtonRequest	Floor No.	integer	1
floorButtonRequest	Direction	string	2
offFloorLamp	Floor No.	integer	1
onOffDirectionLamp	Floor No.	integer	1
onOffDirectionLamp	Direction	string	2
onOffDirectionLamp	Elevator No.	integer	3

Table 8.4: Floor Successor Operation Table

Operation	Successor	
	Operation	Component
floorButtonRequest	serviceRequest	Scheduler
offFloorLamp	floorLampOff	Floor Lamp External Device
onOffDirectionLamp	directionLampOnOff	Direction Lamp External Device

Table 8.5: Floor Function Status Table

Operation	Status
serviceRequest	Unblock
floorLampOff	Unblock
directionLampOnOff	Unblock

8.4.1.2 Floor Monitor Object

The Floor Monitor Object provides the Floor Operation Statechart Table (Table 8.6) and Floor Operation Statechart Information Table (Table 8.7).

Table 8.6: Floor Operation Statechart Table

Operation	State	Event	Action	Next State
floorButtonRequest	Idle	floorButtonRequestArrived		Calling Floor Button Request
floorButtonRequest	Calling Floor Button Request	floorButtonRequestCalled	Start Timing	Processing Floor Button Request
floorButtonRequest	Processing Floor Button Request	serviceRequestArrived	End Timing	Calling Service Request
floorButtonRequest	Calling Service Request	serviceRequestCalled		Idle
offFloorLamp	Idle	offFloorLampArrived		Calling Off Floor Lamp
offFloorLamp	Calling Off Floor Lamp	offFloorLampCalled	Start Timing	Processing Off Floor Lamp
offFloorLamp	Processing Off Floor Lamp	floorLampOffArrived	End Timing	Calling Floor Lamp Off
offFloorLamp	Calling Floor Lamp Off	floorLampOffCalled		Idle
onOffDirectionLamp	Idle	onOffDirectionLampArrived		Calling On Off Direction Lamp
onOffDirectionLamp	Calling On Off Direction Lamp	onOffDirectionLampCalled	Start Timing	Processing On Off Direction Lamp
onOffDirectionLamp	Processing On Off Direction Lamp	directionLampOnOffArrived	End Timing	Calling Direction Lamp On Off
onOffDirectionLamp	Calling Direction Lamp On Off	directionLampOnOffCalled		Idle

Table 8.7: Floor Operation Statechart Information Table

Operation	Current State	Maximum Operation Time	Start Time	End Time
floorButtonRequest	Idle	1000		
offFloorLamp	Idle	1000		
onOffDirectionLamp	Idle	1000		

8.4.1.3 Floor Wrapper Controller Object

The Floor Wrapper Controller object provides the Floor Operation Criticality Table (Table 8.8).

Table 8.8: Floor Operation Criticality Table

Operation	Criticality
floorButtonRequest	Critical
offFloorLamp	Non-critical
onOffDirectionLamp	Non-critical

8.4.1.4 Floor Reconfiguration Manager Object

The Floor Reconfiguration Manager object has been implemented by means of Floor Reconfiguration Plan Generator and Floor Reconfiguration Plan Executor objects. Both objects have access to the Floor Operation Reconfiguration Status Table (Table 8.9) whereas the Floor Reconfiguration Plan Generator has access to the Floor Reconfiguration Table (Table 8.10).

Table 8.9: Floor Operation Reconfiguration Status Table

Operation	Status
floorButtonRequest	Unblock
offFloorLamp	Unblock
onOffDirectionLamp	Unblock
serviceRequest	Unblock
closeDoor	Unblock
openDoor	Unblock

Table 8.10: Floor Reconfiguration Table

Function	Callee			Caller	
	Function	Function Block	Component	Function	Component
floorButtonRequest	serviceRequest		Scheduler		
offFloorLamp				openDoor	Elevator 1
offFloorLamp				openDoor	Elevator 2
offFloorLamp				openDoor	Elevator 3
on/OffDirectionLamp				closeDoor	Elevator 1
on/OffDirectionLamp				openDoor	Elevator 1
on/OffDirectionLamp				closeDoor	Elevator 2
on/OffDirectionLamp				openDoor	Elevator 2
on/OffDirectionLamp				closeDoor	Elevator 3
on/OffDirectionLamp				openDoor	Elevator 3

8.4.1.5 Floor Repair Manager Object

The Floor Repair Manager object provides the Floor Repair Techniques Table (Table 8.11).

Table 8.11: Floor Repair Techniques Table

Technique	Status
Re-Install component	Non-Used
Replace component	Non-Used

8.4.2 Scheduler COTS Component Wrapper

The Scheduler COTS component wrapper self-manages the Scheduler COTS component using the application independent algorithms presented in chapter 7 and the application-dependent information is described in the following sections. The

application-dependent information has been implemented by means of tables and these tables have been assigned to the Scheduler COTS component wrapper's objects.

8.4.2.1 Scheduler Modified Interface Object

The Scheduler Modified Interface object provides the Scheduler Operation Status Table (Table 8.12), the Scheduler Operation Request Parameter Table (Table 8.13), the Scheduler Operation Return Type Table (Table 8.14), the Scheduler Successor Operation Table (Table 8.15), and the Scheduler Function Status Table (Table 8.16).

Table 8.12: Scheduler Operation Status Table

Operation	Status
serviceRequest	Unblock
updateStatusPlan	Unblock
updateElevatorStatus	Unblock

Table 8.13: Scheduler Operation Request Parameter Table

Operation	Parameter	Type	Order
serviceRequest	Floor No.	integer	1
serviceRequest	Direction	string	2
updateStatusPlan	Floor 1 No.	integer	1
updateStatusPlan	Status 1	string	2
updateStatusPlan	Floor 2 No.	integer	3
updateStatusPlan	Status 2	string	4
updateStatusPlan	Elevator No.	integer	5
updateElevatorStatus	Floor No.	integer	1
updateElevatorStatus	Status	string	2
updateElevatorStatus	Elevator No.	integer	3

Table 8.14: Scheduler Operation Return Type Table

Operation	Parameter	Type	Order
serviceRequest	Floor No.	integer	1
serviceRequest	Direction	string	2
serviceRequest	Elevator No.	integer	3
updateStatusPlan	Acknowledge	string	2
updateElevatorStatus	Acknowledge	string	2

Table 8.15: Scheduler Successor Operation Table

Operation	Successor	
	Operation	Component
serviceRequest	scheduleRequest	Elevator
updateStatusPlan	acknowledge	
updateElevatorStatus	acknowledge	

Table 8.16: Scheduler Function Status Table

Operation	Status
scheduleRequest1	Unblock
scheduleRequest2	Unblock
scheduleRequest3	Unblock

8.4.2.2 Scheduler Monitor Object

The Scheduler Monitor Object provides the Scheduler Operation Statechart Table (Table 8.17) and Scheduler Operation Statechart Information Table (Table 8.18).

Table 8.17: Scheduler Operation Statechart Table

Operation	State	Event	Action	Next State
serviceRequest	Idle	serviceRequestArrived		Calling Service Request
serviceRequest	Calling Service Request	serviceRequestCalled	Start Timing	Processing Service Request
serviceRequest	Processing Service Request	scheduleRequestArrived	End Timing	Calling Schedule Request
serviceRequest	Calling Schedule Request	scheduleRequestCalled		Idle
updateStatusPlan	Idle	updateStatusPlanArrived		Calling Update Status Plan
updateStatusPlan	Calling Update Status Plan	updateStatusPlanCalled	Start Timing	Processing Update Status Plan
updateStatusPlan	Processing Update Status Plan	acknowledgeArrived	End Timing	Idle
updateElevatorStatus	Idle	updateElevatorStatusArrived		Calling Update Elevator Status
updateElevatorStatus	Calling Update Elevator Status	updateElevatorStatusCalled	Start Timing	Processing Update Elevator Status
updateElevatorStatus	Processing Update Elevator Status	acknowledgeArrived	End Timing	Idle

Table 8.18: Scheduler Operation Statechart Information Table

Operation	Current State	Maximum Operation Time	Start Time	End Time
serviceRequest	Idle	5000		
updateStatusPlan	Idle	10000		
updateElevatorStatus	Idle	10000		

8.4.2.3 Scheduler Wrapper Controller Object

The Scheduler Wrapper Controller object provides the Scheduler Operation Criticality Table (Table 8.19).

Table 8.19: Scheduler Operation Criticality Table

Operation	Criticality
serviceRequest	Critical
updateStatusPlan	Critical
updateElevatorStatus	Critical

8.4.2.4 Scheduler Reconfiguration Manager Object

The Scheduler Reconfiguration Manager object has been implemented by means of Scheduler Reconfiguration Plan Generator and Scheduler Reconfiguration Plan Executor objects. Both objects have access to the Scheduler Operation Reconfiguration Status Table (Table 8.20) whereas the Scheduler Reconfiguration Plan Generator has access to the Scheduler Reconfiguration Table (Table 8.21).

Table 8.20: Scheduler Operation Reconfiguration Status Table

Operation	Status
serviceRequest	Unblock
updateStatusPlan	Unblock
updateElevatorStatus	Unblock
scheduleRequest(elevator 1)	Unblock
scheduleRequest(elevator 2)	Unblock
scheduleRequest(elevator 3)	Unblock
floorButtonRequest	Unblock
elevatorButtonRequest	Unblock
scheduleRequest	Unblock
sensorInput	Unblock

Table 8.21: Scheduler Reconfiguration Table

Function	Callee			Caller	
	Function	Function Block	Component	Function	Component
serviceRequest		scheduleRequest1	Elevator 1	floorButtonRequest	Floor
		scheduleRequest2	Elevator 2		
		scheduleRequest3	Elevator 3		
updateStatusPlan				scheduleRequest	Elevator 1
updateStatusPlan				elevatorButtonRequest	Elevator 1
updateStatusPlan				scheduleRequest	Elevator 2
updateStatusPlan				elevatorButtonRequest	Elevator 2
updateStatusPlan				scheduleRequest	Elevator 3
updateStatusPlan				elevatorButtonRequest	Elevator 3
updateElevatorStatus				sensorInput	Elevator 1
updateElevatorStatus				sensorInput	Elevator 2
updateElevatorStatus				sensorInput	Elevator 3

8.4.2.5 Scheduler Repair Manager Object

The Scheduler Repair Manager object provides the Scheduler Repair Techniques Table (Table 8.22).

Table 8.22: Scheduler Repair Techniques Table

Technique	Status
Re-Install component	Non-Used
Replace component	Non-Used

8.4.3 Elevator COTS Component Wrapper

The Elevator COTS component wrapper self-manages the Elevator COTS component using the application independent algorithms presented in chapter 7 and the application-dependent information as described in the following sections. The

application-dependent information has been implemented by means of tables and these tables have been assigned to the Elevator COTS component wrapper's objects.

8.4.3.1 Elevator Modified Interface Object

The Elevator Modified Interface object provides the Elevator Operation Status Table (Table 8.23), the Elevator Operation Request Parameter Table (Table 8.24), the Elevator Operation Return Type Table (Table 8.25), the Elevator Successor Operation Table (Table 8.26), and the Elevator Function Status Table (Table 8.27).

Table 8.23: Elevator Operation Status Table

Operation	Status
scheduleRequest	Unblock
sensorInput	Unblock
runMotor	Unblock
stopMotor	Unblock
openDoor	Unblock
closeDoor	Unblock
elevatorButtonRequest	Unblock

Table 8.24: Elevator Operation Request Parameter Table

Operation	Parameter	Type	Order
scheduleRequest	Floor No.	integer	1
scheduleRequest	Direction	string	2
scheduleRequest	Elevator No.	integer	3
sensorInput	Floor No.	integer	1
sensorInput	Status	string	2
sensorInput	Elevator No.	integer	3
runMotor			
stopMotor			
openDoor	Floor No.	integer	1
openDoor	Status	string	2
openDoor	Elevator No.	integer	3
closeDoor	Floor No.	integer	1
closeDoor	Status	string	2
closeDoor	Elevator No.	integer	3
elevatorButtonRequest	Floor No.	integer	1
elevatorButtonRequest	Elevator No.	integer	3

Table 8.25: Elevator Operation Return Type Table

Operation	Parameter	Type	Order
scheduleRequest	Floor 1 No.	integer	1
scheduleRequest	Status 1	string	2
scheduleRequest	Floor 2 No.	integer	3
scheduleRequest	Status 2	string	4
scheduleRequest	Elevator No.	integer	5
sensorInput	Floor No.	integer	1
sensorInput	Status	string	2
sensorInput	Elevator No.	integer	3
runMotor	Action	string	2
stopMotor	Action	string	2
openDoor	Floor No.	integer	1
openDoor	Action	string	2
openDoor	Elevator No.	integer	3
closeDoor	Floor No.	integer	1
closeDoor	Action	string	2
closeDoor	Elevator No.	integer	3
elevatorButtonRequest	Floor 1 No.	integer	1
elevatorButtonRequest	Direction 1	string	2
elevatorButtonRequest	Floor 2 No.	integer	3
elevatorButtonRequest	Direction 2	string	4
elevatorButtonRequest	Elevator No.	integer	5

Table 8.26: Elevator Successor Operation Table

Operation	Successor	
	Operation	Component
scheduleRequest	updateStatusPlan	Scheduler
sensorInput	updateElevatorStatus	Scheduler
runMotor	motorRun	Motor External Device
stopMotor	motorStop	Motor External Device
openDoor	doorOpen	Door External Device
openDoor	offFloorLamp	Floor
openDoor	onOffDirectionLamp	Floor
closeDoor	doorClose	Door External Device
closeDoor	onOffDirectionLamp	Floor
elevatorButtonRequest	updateStatusPlan	Scheduler

Table 8.27: Elevator Function Status Table

Operation	Status
updateStatusPlan	Unblock
updateElevatorStatus	Unblock
offFloorLamp	Unblock
onOffDirectionLamp	Unblock
motorRun	Unblock
motorStop	Unblock
doorOpen	Unblock
doorClose	Unblock

8.4.3.2 Elevator Monitor Object

The Elevator Monitor Object provides the Elevator Operation Statechart Table (Table 8.28) and Elevator Operation Statechart Information Table (Table 8.29).

Table 8.28: Elevator Operation Statechart Table

Operation	State	Event	Action	Next State
scheduleRequest	Idle	scheduleRequestArrived		Calling Schedule Request
scheduleRequest	Calling Schedule Request	scheduleRequestCalled	Start Timing	Processing Schedule Request
scheduleRequest	Processing Schedule Request	updateStatusPlanArrived	End Timing	Calling Update Status Plan
scheduleRequest	Calling Update Status Plan	updateStatusPlanCalled		Idle
sensorInput	Idle	sensorInputArrived		Calling Sensor Input
sensorInput	Calling Sensor Input	sensorInputCalled	Start Timing	Processing Sensor Input
sensorInput	Processing Sensor Input	updateElevatorStatusArrived	End Timing	Calling Update Elevator Status
sensorInput	Calling Update Elevator Status	updateElevatorStatusCalled		Idle
runMotor	Idle	runMotorArrived		Calling Run Motor
runMotor	Calling Run Motor	runMotorCalled	Start Timing	Processing Run Motor
runMotor	Processing Run Motor	motorRunArrived	End Timing	Calling Motor Run
runMotor	Calling Motor Run	motorRunCalled		Idle
stopMotor	Idle	stopMotorArrived		Calling Stop Motor
stopMotor	Calling Stop Motor	stopMotorCalled	Start Timing	Processing Stop Motor
stopMotor	Processing Stop Motor	motorStopArrived	End Timing	Calling Motor Stop
stopMotor	Calling Motor Stop	motorStopCalled		Idle
openDoor	Idle	openDoorArrived		Calling Open Door
openDoor	Calling Open Door	openDoorCalled	Start Timing	Processing Open Door
openDoor	Processing Open Door	doorOpenArrived	End Timing	Calling Door Open
openDoor	Calling Door Open	doorOpenCalled		Idle
closeDoor	Idle	closeDoorArrived		Calling Close Door
closeDoor	Calling Close Door	closeDoorCalled	Start Timing	Processing Close Door
closeDoor	Processing Close Door	doorCloseArrived	End Timing	Calling Close Door
closeDoor	Calling Door Close	doorCloseCalled		Idle
elevatorButtonRequest	Idle	elevatorButtonRequestArrived		Calling Elevator Button Request
elevatorButtonRequest	Calling Elevator Button Request	elevatorButtonRequestCalled	Start Timing	Processing Elevator Button Request
elevatorButtonRequest	Processing Elevator Button Request	updateStatusPlanArrived	End Timing	Calling Update Status Plan
elevatorButtonRequest	Calling Update Status Plan	updateStatusPlanCalled		Idle

Table 8.29: Elevator Operation Statechart Information Table

Operation	Current State	Maximum Operation Time	Start Time	End Time
scheduleRequest	Idle	10000		
sensorInput	Idle	10000		
runMotor	Idle	10000		
stopMotor	Idle	10000		
openDoor	Idle	10000		
closeDoor	Idle	10000		
elevatorButtonRequest	Idle	10000		

8.4.3.3 Elevator Wrapper Controller Object

The Elevator Wrapper Controller object provides the Elevator Operation Criticality Table (Table 8.30).

Table 8.30: Elevator Operation Criticality Table

Operation	Criticality
scheduleRequest	Critical
sensorInput	Critical
runMotor	Critical
stopMotor	Critical
openDoor	Critical
closeDoor	Critical
elevatorButtonRequest	Critical

8.4.3.4 Elevator Reconfiguration Manager Object

The Elevator Reconfiguration Manager object has been implemented by means of Elevator Reconfiguration Plan Generator and Elevator Reconfiguration Plan Executor objects. Both objects have access to the Elevator Operation Reconfiguration Status Table (Table 8.31) whereas the Elevator Reconfiguration Plan Generator has access to the Elevator Reconfiguration Table (Table 8.32).

Table 8.31: Elevator Operation Reconfiguration Status Table

Operation	Status
scheduleRequest	Unblock
sensorInput	Unblock
runMotor	Unblock
stopMotor	Unblock
openDoor	Unblock
closeDoor	Unblock
elevatorButtonRequest	Unblock
serviceRequest	Unblock
updateStatusPlan	Unblock
updateElevatorStatus	Unblock
offFloorLamp	Unblock
onOffDirectionLamp	Unblock

Table 8.32: Elevator Reconfiguration Table

Function	Callee			Caller	
	Function	Function Block	Component	Function	Component
scheduleRequest	updateStatusPlan		Scheduler	serviceRequest	Scheduler
sensorInput	updateElevatorStatus		Scheduler		
runMotor					
stopMotor					
openDoor		offFloorLamp	Floor		
		onOffDirectionLamp	Floor		
closeDoor		onOffDirectionLamp	Floor		
elevatorButtonRequest	updateStatusPlan		Scheduler		

8.4.3.5 Elevator Repair Manager Object

The Elevator Repair Manager object provides the Elevator Repair Techniques Table (Table 8.33).

Table 8.33: Elevator Repair Techniques Table

Technique	Status
Re-Install component	Non-Used
Replace component	Non-Used

8.5 Validation of the Distributed Elevator System

This section presents the validation of the mechanisms provided by a *COTS component wrapper* for self-managing COTS components. The application-independent algorithms of a *COTS component wrapper* are validated to determine if the algorithms can detect, reconfigure, and recover anomalies of COTS components correctly.

8.5.1 Hardware and Software Specifications

The test cases used to validate the Detection Mechanism (chapter 4) the Reconfiguration Mechanism (chapter 5) and the Repair Mechanism (chapter 6) were executed by a computer with the following hardware and software specifications:

Hardware:

- Processor: Intel® Core™2 CPU – T5500
- Speed: 1.66 GHz – 1.67 GHz.
- Memory (RAM): 1.00 GB
- System type: 32-bit Operating System

Software:

- Operative System: Windows Vista™ Home Premium with Service Pack 1
- Development Environment: Eclipse Platform Version 3.2.0
- Java™ SE Runtime Environment: 1.6.0-rc

8.5.2 Validation of the Detection Mechanism's Algorithms

The Detection Mechanism (chapter 4) was validated using the first case-study, the distributed Elevator system (Gomaa 2000) with multiple elevators. The validation of the

Detection Mechanism involves the procedure of the testing, the related objects involved in the *COTS component wrapper*, the application-independent algorithms provided by each object, the test cases performed, and the result of those test cases.

8.5.2.1 Testing Procedure for Detection Mechanism

- 1) Execute the system three times to obtain the execution times for each function in the COTS component.
- 2) Establish the maximum execution time for each function in the Operation Statechart Information Table as the average of the three execution times plus a 50%.
- 3) Check if the COTS Modified Interface object extracts correctly the function name from the parameters received.
- 4) Verify if the COTS Modified Interface object extracts and shows to the console the function's arguments using the Operation Request Parameter Table.
- 5) Validate if the COTS Modified Interface object checks the status of each function arrived in the Operation Status Table and allows the function to be invoked from the COTS component only when the status is 'unblock'. Otherwise, the COTS Modified Interface object only displays a message indicating that the function is blocked.
- 6) Check if the COTS Modified Interface object sends the functionName + 'Arrived' event to the COTS Monitor object, and if the function's statechart transits to the corresponding state using the Operation Statechart Table and the Operation Statechart Information Table.

- 7) Verify if the COTS Modified Interface object calls the function from the COTS component with the corresponding arguments.
- 8) Check if the COTS Modified Interface object sends the functionName + 'Called' event to the COTS Monitor object, and if the function's statechart transits to the corresponding state using the Operation Statechart Table and the Operation Statechart Information Table.
- 9) Validate if the COTS Monitor object starts timing the function based on the event received using the maximum execution time established in the Operation Statechart Information Table.
- 10) Verify if the COTS Modified Interface object receives the reply from the COTS component and its arguments are same as defined in the Operation Return Type Table.
- 11) Check if the COTS Modified Interface object identifies correctly the successor function in the Successor Operation Table.
- 12) Validate if the COTS Modified Interface object sends the successorFunction + 'Arrived' event to the COTS Monitor object, and if the execution time for that function is less or equal to the maximum execution time recorded in the Statechart Operation Table. If this condition is satisfied, check if the function's statechart transits to the corresponding state using the Operation Statechart Table and the Operation Statechart Information Table. If this condition is not satisfied, check if the COTS Monitor object detects the error and communicates the function name to the Wrapper Controller object.

- 13) If the successor function is provided by a COTS component, verify: if the COTS Modified Interface object calls the successor function, if the successorFunction + 'Called' event is sent to the COTS Monitor object by the COTS Modified Interface object, and if the function's statechart transits to the Idle state.
- 14) If the successor function is just an acknowledgment, verify that the function's statechart transits to the Idle state based on the successorFunction + 'Arrived' event sent by the COTS Modified Interface object to the COTS Monitor object in step 12.
- 15) If the COTS Modified Interface object does not receive the reply from the COTS component due to an error in the COTS component (step 10), and the maximum execution time has expired, check if the COTS Monitor object detects the error and sends the function name to the Wrapper Controller object.

8.5.2.2 Related Objects in COTS Component Wrapper

The Detection Mechanism is supported by the following COTS component wrapper objects: COTS Modified Interface, COTS Monitor, and Wrapper Controller objects.

8.5.2.3 Operations Provided by each Object in the COTS Component Wrapper

Table 8.34 shows the application-independent algorithms provided by each object of the COTS component wrapper involved in the Detection Mechanism.

Table 8.34: Application-Independent Algorithms for Detection Mechanism

Algorithm	COTS component wrapper object
OperationRequest(in parameters)	COTS Modified Interface object
ModifiedInterfaceNotification(in operation, event)	COTS Monitor object
StartTime(in operation, time)	COTS Monitor object
MonitorTimeNotification(in operation)	COTS Monitor object
MalfunctionOperation(in operation)	Wrapper Controller object

8.5.2.4 Testing

To validate the Detection Mechanism in the *Floor COTS component*, six text cases were used, two for each function provided by the Floor COTS component: Normal case and Fault or error case.

1) **floorButtonRequest** function in Floor COTS component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Floor COTS component preventing the Floor COTS component to reply the floor number and direction of the elevator to the COTS Modified Interface object. In the FloorCOTS.java class, the line 34 was eliminated.

2) **offFloorLamp** function in Floor COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Floor COTS component preventing the Floor COTS component to reply the floor number whose lamp needs to be powered off to the COTS Modified Interface object. In the FloorCOTS.java class, the line 41 was eliminated.

3) **onOffDirectionLamp** function in Floor COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Floor COTS component preventing the Floor COTS component to reply the floor number, direction of the elevator, and elevator number to the COTS Modified Interface object. In the FloorCOTS.java class, the line 48 was eliminated.

The validation of the Detection Mechanism in the Scheduler COTS component and Elevator COTS component and the test results are located in appendix B.

8.5.2.5 Test Result

The six tests performed on the *Floor COTS component* provide the following results:

1) **floorButtonRequest** function in Floor COTS component

- a. Normal case: the function performed its activity successfully and the COTS Modified Interface object called the corresponding serviceRequest function from the Scheduler COTS component wrapper.
- b. Fault or error case: the COTS Monitor object detected the error in the floorButtonRequest function, and communicated the malfunction to the Wrapper Controller object who initiated the reconfiguration process.

2) **offFloorLamp** function in Floor COTS Component

- a. Normal case: the function performed its activity successfully and the COTS Modified Interface object called the corresponding floorLampOff function from the Floor Lamp External Device.
- b. Fault or error case: the COTS Monitor object detected the error in the offFloorLamp function, and communicated the malfunction to the Wrapper Controller object who initiated the reconfiguration process.

3) **onOffDirectionLamp** function in Floor COTS Component

- a. Normal case: the function performed its activity successfully and the COTS Modified Interface object called the corresponding directionLampOnOff function from the Direction Lamp External Device.
- b. Fault or error case: the COTS Monitor object detected the error in the onOffDirectionLamp function, and communicated the malfunction to the Wrapper Controller object who initiated the reconfiguration process.

8.5.3 Validation of the Reconfiguration Mechanism's Algorithms

The Reconfiguration Mechanism (chapter 5) was also validated using the first case-study, the distributed Elevator system (Gomaa 2000) with multiple elevators. The validation of the Reconfiguration Mechanism involves the procedure of the testing, the related objects involved in the *COTS component wrapper*, the application-independent

algorithms provided by each object, the test cases performed, and the result of those test cases.

8.5.3.1 Testing Procedure for Reconfiguration Mechanism

- 1) Check if the Wrapper Controller object receives the malfunction function from the COTS Monitor object and identifies the criticality of the malfunction function successfully using the Operation Criticality Table.
- 2) Validate if the Wrapper Controller object sends the malfunction function with its corresponding action to the Reconfiguration Plan Generator object.
- 3) Verify if the Reconfiguration Plan Generator object generates the Reconfiguration Plan for the malfunction function using its Operation Reconfiguration Status Table and Reconfiguration Table.
- 4) Check if the Reconfiguration Plan Generator object sends the Reconfiguration Plan to the Reconfiguration Plan Executor object.
- 5) Validate if the Reconfiguration Plan Executor object executes each one of the lines established in the Reconfiguration Plan.
- 6) Verify if the Reconfiguration Plan Executor object updates the status of each function specified in the Reconfiguration Plan in the Operation Reconfiguration Status Table.
- 7) In the case where a function is provided by the COTS component wrapped, check if the Reconfiguration Plan Executor object sends the message to the COTS Modified Interface object to perform the action specified in the Reconfiguration Plan. In addition, validate if the COTS Modified Interface object

notifies to the Reconfiguration Plan Executor object when the action has been updated in its Operation Status Table.

- 8) In the case where a function has non dependent functions in other COTS components, checks if the Reconfiguration Plan Executor object shows the action to the console.
- 9) In the case where a function is provided by other COTS components, verify if the Reconfiguration Plan Executor object sends the message to the Wrapper Controller object to reconfigure other dependents COTS components with the action specified in the Reconfiguration Plan.
- 10) In the case of the reconfiguration after a repair has been performed, check if the Reconfiguration Plan Executor object sends the message to the COTS Monitor object to reinitialize the statechart of the malfunction operation. In addition, validate if the COTS Monitor object notifies to the Reconfiguration Plan Executor object when the statechart is reinitialized.
- 11) When the Reconfiguration Plan has been executed, verify if the Reconfiguration Plan Executor object notifies the finalization of the reconfiguration process to the Wrapper Controller object.
- 12) Check if the Wrapper Controller object initiates the repair process when the component should be repaired.
- 13) Validate if the Wrapper Controller object receives the malfunction from the Wrapper Controller object in other COTS components, and performs the reconfiguration process in the COTS component wrapped.

8.5.3.2 Related Objects in COTS Component Wrapper

The Reconfiguration Mechanism is supported by the following COTS component wrapper objects: Wrapper Controller, Reconfiguration Plan Generator, Reconfiguration Plan Executor, COTS Modified Interface, and COTS Monitor objects.

8.5.3.3 Operations Provided by each Object in the COTS Component Wrapper

Table 8.35 shows the application-independent algorithms provided by each object of the COTS component wrapper involved in the Reconfiguration Mechanism.

Table 8.35: Application-Independent Algorithms for Reconfiguration Mechanism

Algorithm	COTS component wrapper object
MalfunctionOperation(in operation)	Wrapper Controller object
GenerateReconfigurationPlan(in operation, object, action)	Reconfiguration Plan Generator object
ExecuteReconfigurationPlan(in reconfiguration plan)	Reconfiguration Plan Executor object
UpdateOperationStatus(in operation, action)	COTS Modified Interface object
ModifiedInterfaceNotification(in notification)	Reconfiguration Plan Executor object
ReinitializeStatechart(in operation)	COTS Monitor object
MonitorNotification(in notification)	Reconfiguration Plan Executor object
ReconfigurationExecutorNotification(in operation, object, action, notification)	Wrapper Controller object
MalfunctionOperationOtherComponents(in operation, object, action)	Wrapper Controller object

8.5.3.4 Testing

To validate the Reconfiguration Mechanism in the *Floor COTS component*, three test cases were used, one for each function provided by the Floor COTS component. Each test case contains the following information:

- Criticality of the function.

- Reconfiguration plan against the anomalous function. The reconfiguration plan contains information such as function name, object providing the function, and the action to be performed.

In the case where an anomalous function contains dependent functions in other COTS components, the reconfiguration plans of those dependent COTS component are depicted as well.

1) **floorButtonRequest** function in Floor COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Floor COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Table 8.36 shows the floor reconfiguration plan against the anomalous floorButtonRequest function.

Table 8.36: Floor Reconfiguration Plan for Floor Button Request Function

Line No.	Operation	Object	Action
1	floorButtonRequest		
2	allOperations	Floor	Repair Block
3	floorButtonRequest		Repair Block
4	floorButtonRequest	Floor	Notify Console
5	offFloorLamp		Repair Block
6	offFloorLamp	Elevator1	Repair Block
7	openDoor		Dependency Block
8	offFloorLamp	Elevator2	Repair Block
9	openDoor		Dependency Block
10	offFloorLamp	Elevator3	Repair Block
11	openDoor		Dependency Block
12	onOffDirectionLamp		Repair Block
13	onOffDirectionLamp	Elevator1	Repair Block
14	closeDoor		Dependency Block
15	openDoor		Dependency Block
16	onOffDirectionLamp	Elevator2	Repair Block
17	closeDoor		Dependency Block
18	openDoor		Dependency Block
19	onOffDirectionLamp	Elevator3	Repair Block
20	openDoor		Dependency Block
21	closeDoor		Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1:* This is the name of the function that is malfunction, so the system can identify the reconfiguration plan.
- *Line 2:* This function is critical; therefore, all the functions in Floor COTS component must be blocked as ‘Repair Block’. This line is used to communicate only once to the COTS Modified Interface object to repair all the operations provided by the Floor COTS component.
- *Lines 3, 5, 7, 9, 11, 12, 14, 15, 17, 18, 20, and 21:* These lines are used to change the status of the functions in the Floor Operation

Reconfiguration Status Table in the Reconfiguration Manager object.

- *Line 4*: This function has no dependent operations in other components and because it is critical, a message should be display to the console.
- *Lines 6, 8, 10, 13, 16, and 19*: These functions have dependent operations in Elevator COTS components; therefore, those functions have to be blocked as ‘Repair Block’. These lines are used for the Reconfiguration Plan Executor object to send the malfunction notifications to the Elevator Wrapper Controller objects.

c. Reconfiguration in the Elevator COTS Component: The Elevator Wrapper Controllers receives two functions that are malfunctions. Those reconfiguration plans are as follows:

- offFloorLamp function: Table 8.37 shows the Elevator Reconfiguration Plan for the Off Floor Lamp malfunction function.

Table 8.37: Elevator Reconfiguration Plan for Off Floor Lamp Function

Line No.	Operation	Object	Action
1	offFloorLamp		
2	offFloorLamp		Repair Block
3	offFloorLamp	Elevator	Function Block

The explication of each line of the reconfiguration plan follows:

- *Line 1:* This is the function name that is malfunction, so the system can identify the reconfiguration plan.
- *Line 2:* This line is used to change the status of the function in the Elevator Reconfiguration Operation Reconfiguration Status Table in the Reconfiguration Manager object.
- *Line 3:* This line is used to communicate to the COTS Modified Interface to ‘Function Block’ the function. This kind of blocking allows the Elevator COTS component to work normally, with the only difference that the COTS Modified Interface is not going to call the offFloorLamp function from the Floor Modified Interface.
- onOffDirectionLamp function: Table 8.38 shows the Elevator Reconfiguration Plan for the On Off Direction Lamp malfunction function.

Table 8.38: Elevator Reconfiguration Plan for On Off Direction Lamp Function

Line No.	Operation	Object	Action
1	onOffDirectionLamp		
2	onOffDirectionLamp		Repair Block
3	onOffDirectionLamp	Elevator	Function Block
4	onOffDirectionLamp		Repair Block
5	onOffDirectionLamp	Elevator	Function Block

The explication of each line of the reconfiguration plan follows:

- *Line 1:* This is the function name that is malfunction, so the system can identify the reconfiguration plan.

- *Lines 2 and 4:* These lines are used to change the status of the function in the Elevator Operation Reconfiguration Status Table in the Reconfiguration Manager object.
- *Lines 3 and 5:* These lines are used to communicate to the COTS Modified Interface to 'Function Block' the operation. This kind of blocking allows the Elevator COTS component to work normally, with the only difference that the COTS Modified Interface is not going to call the onOffDirectionLamp function from the Floor Modified Interface.

The reconfiguration plan shows the same lines (2 and 4; 3 and 5) two times because there are two functions in the Elevator COTS component that are dependent of the onOffDirectionLamp function.

- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plan before repair showed above. The only difference is that the action to be taken is 'Unblock' instead of 'Repair Block', 'Malfunction Block', 'Dependency Block', or 'Function Block'.

2) **offFloorLamp** function in Floor COTS component

- a. Criticality of the function: this function is non critical for the system; therefore, the Floor COTS component provides degraded service. This function can be repair during the off-line maintenance period.
- b. Reconfiguration Plan before repair: The Table 8.39 shows the Floor Reconfiguration Plan for the Off Floor Lamp malfunction function.

Table 8.39: Floor Reconfiguration Plan for Off Floor Lamp Function

Line No.	Operation	Object	Action
1	offFloorLamp		
2	offFloorLamp		Malfunction Block
3	offFloorLamp	Floor	Malfunction Block
4	offFloorLamp		Malfunction Block
5	offFloorLamp	Elevator1	Malfunction Block
6	openDoor		Dependency Block
7	offFloorLamp	Elevator2	Malfunction Block
8	openDoor		Dependency Block
9	offFloorLamp	Elevator3	Malfunction Block
10	openDoor		Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1:* This is the function name that is malfunction, so the system can identify the reconfiguration plan.
- *Line 3:* This line is used to communicate to the COTS Modified Interface to block as malfunction the offFloorLmap function.
- *Lines 2, 4, 6, 8, and 10:* These lines are used to change the status of the functions in the Floor Operation Reconfiguration Status Table in the Reconfiguration Manager object.
- *Lines 5, 7, and 9:* These functions have dependent functions in Elevator COTS components; therefore, those operations have to be blocked as ‘Malfunction Block’. These lines are used for the Reconfiguration Manager to send the malfunction notifications to the Elevator Wrapper Controllers.

- c. Reconfiguration in the Elevator COTS components: The Elevator Wrapper Controllers receive the malfunction function generating the same reconfiguration plan depicted in Table 8.37 with the only difference that in this case the action informed is ‘malfunction block’ instead of ‘repair block’.

3) **onOffDirectionLamp** function in Floor COTS component

- a. Criticality of the function: this function is non critical for the system; therefore, the Floor COTS component provides degraded service. This function can be repair during the off-line maintenance period.
- b. Reconfiguration Plan before repair: Table 8.40 shows the Floor Reconfiguration Plan for the On Off Direction Lamp malfunction function.

Table 8.40: Floor Reconfiguration Plan for On Off Direction Lamp Function.

Line No.	Operation	Object	Action
1	onOffDirectionLamp		
2	onOffDirectionLamp		Malfunction Block
3	onOffDirectionLamp	Floor	Malfunction Block
4	onOffDirectionLamp		Malfunction Block
5	onOffDirectionLamp	Elevator1	Malfunction Block
6	closeDoor		Dependency Block
7	openDoor		Dependency Block
8	onOffDirectionLamp	Elevator2	Malfunction Block
9	closeDoor		Dependency Block
10	openDoor		Dependency Block
11	onOffDirectionLamp	Elevator3	Malfunction Block
12	closeDoor		Dependency Block
13	openDoor		Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1*: This is the function name that is malfunction, so the system can identify the reconfiguration plan.
 - *Line 3*: This line is used to communicate to the COTS Modified Interface to block as malfunction the offFloorLmap function.
 - *Lines 2, 4, 6, 7, 9, 10, 12, and 13*: These lines are used to change the status of the operations in the Floor Operation Reconfiguration Status Table in the Reconfiguration Manager object.
 - *Lines 5, 8, and 11*: These functions have dependent operations in Elevator COTS components; therefore, those functions have to be blocked as ‘Malfunction Block’. These lines are used for the Reconfiguration Manager to send the malfunction notifications to the Elevator Wrapper Controllers.
- c. Reconfiguration in the Elevator COTS components: The Elevator Wrapper Controllers receive the malfunction function generating the same reconfiguration plan depicted in Table 8.38 with the only difference that in this case the action informed is ‘malfunction block’ instead of ‘repair block’.

The validation of the Reconfiguration Mechanism in the Scheduler COTS component and Elevator COTS component and the test results are located in appendix B.

8.5.3.5 Test Results

The three tests performed on the *Floor COTS component* provide the following results:

- a) Criticality of the operation: the Floor Wrapper Controller had successfully identified the criticality of each operation selecting ‘Critical’ for the test case 1 and ‘Non-Critical’ for the test cases 2 and 3.
- b) Reconfiguration Plan before repair: the Floor Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test cases 1 to 3 (Tables 8.36, 8.37, 8.38, 8.39, and 8.40).
- c) Reconfiguration Plan after repair: The Floor Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test cases 1 to 3, changing the action to ‘unblock’ (Tables 8.36, 8.37, 8.38, 8.39, and 8.40).

8.5.4 Validation of the Repair Mechanism’s Algorithms

The Repair Mechanism (chapter 6) was also validated using the first case-study, the distributed Elevator system (Gomaa 2000) with multiple elevators. The validation of the Repair Mechanism involves the procedure of the testing, the related objects involved in the *COTS component wrapper*, the application-independent algorithms provided by each object, the test cases performed, and the result of those test cases.

8.5.4.1 Testing Procedure for Repair Mechanism

- 1) Check if the Repair Manager receives the notification to start the repair process from the Wrapper Controller.
- 2) Validate if the Repair Manager choose the first repair technique that has not been used yet using the Repair Techniques Table.
- 3) Verify if the Repair Manager performs the repair technique selected.
- 4) Check if the Repair Manager sends the new object generated according to the technique selected to the COTS Modified Interface.
- 5) Validate if the COTS Modified Interface receives the new object and uses this object as a new COTS component.
- 6) Verify if the COTS Modified Interface sends the notification to the Repair Manager indicating that the object has been received.
- 7) Check if the Repair Manager notifies to the Wrapper Controller that the repair process has been executed, and the reconfiguration after repair has to be executed.

8.5.4.2 Related Objects in COTS Component Wrapper

The Repair Mechanism is supported by the following COTS component wrapper objects: Repair Manager, COTS Modified Interface, and Wrapper Controller objects.

8.5.4.3 Operations Provided by each Object in the COTS Component Wrapper

Table 8.41 shows the application-independent algorithms provided by each object of the COTS component wrapper involved in the Repair Mechanism.

Table 8.41: Application-Independent Algorithms for Repair Mechanism

Algorithm	COTS component wrapper object
StartRepairProcess(in component, notification)	Repair Manager object
RepairManagerNotification(in object, notification)	COTS Modified Interface object
RepairExecuted(in notification)	Repair Manager object
RepairManagerNotification(in notification)	Wrapper Controller object

8.5.4.4 Testing

To validate the Repair Mechanism in the *Floor COTS component*, two test cases were used, one for each technique available to repair the Floor COTS component specified in the Floor Repair Techniques Table (Table 8.11).

- 1) Re-install component: The technique re-installs the same component using a copy of the Floor COTS component kept in a safe place.
- 2) Replace component: The technique selects a variant version of the Floor COTS component as a new Floor COTS component. To validate this test case, the first technique has been marked as ‘used’, so the algorithm will select this second technique.

The validation of the Repair Mechanism in the Scheduler COTS component and Elevator COTS component and the test results are located in appendix B.

8.5.4.5 Test Results

The two tests performed on the *Floor COTS component* provide the following results:

- 1) Re-install component: The Floor Repair Manager object selects and executes this technique to repair the Floor COTS component successfully. The Floor COTS component provides full services using a copy of the same Floor COTS component.
- 2) Replace component: The Floor Repair Manager object selects and executes this second technique successfully. The Floor COTS component provides full services using a variant version of the Floor COTS component.

8.6 Summary

This research has been validated through a distributed elevator system case study. This case study first implemented the *COTS component wrappers* for the Elevator COTS components, the Floor COTS components, and the Scheduler COTS component. In this implementation, the application-dependent information associated with each one of the COTS component involved has been specified by means of tables. Next, the self-management mechanisms have been evaluated. A series of test cases were developed and executed to determine if the Elevator COTS component wrapper, the Floor COTS component wrapper and the Scheduler COTS component wrapper can detect a malfunction of any of their functions. For these test cases, a fault was inserted in each of the functions provided by each of the COTS component to determine if the COTS component wrappers can detect them. Once a fault has been detected, another series of test cases were performed to verify if the whole system is reconfigured against the malfunctions. Finally, a last series of test were performed to verify if the *COTS*

component wrappers can repair the Elevator COTS components, Floor COTS components, and Scheduler COTS component.

CHAPTER 9

DISTRIBUTED FACTORY AUTOMATION SYSTEM: A CASE STUDY

9.1 Introduction

Similar to the distributed elevator system case study in chapter 8, the distributed factory automation system case study was developed to validate the approach for self-managing COTS component based systems. The COTS components were developed first, using the design provided by Gomaa (Gomaa 2000), so the introduction of bias in the system was minimized and the implementation was not adapted to favorite the approach presented in this research. Once the COTS components were developed, the turn was to implement the *COTS component wrappers*.

The software architecture of the distributed factory automation system is developed by structuring the system into one Production Manager COTS Component, one Process Engineer COTS Component, one Receiving Workstation Controller COTS Component, three Line Workstation Controller COTS Components, and one Shipping Workstation Controller COTS Component. In the implementation of the *COTS component wrappers* corresponding to each one of the COTS components, the application-dependent information associated with each of the COTS components involved in the distributed factory automation system is depicted by means of tables. To validate the self-managing mechanisms proposed in this approach, a series of test cases were performed.

This chapter begins by describing the distributed factory automation system used in this case study in Section 9.2. Section 9.3 shows the software architecture of the distributed factory automation system. Section 9.4 presents the implementation of the self-managed distributed factory automation system. Section 9.5 addresses the validation of this research. Section 9.6 summarizes this chapter.

9.2 Problem Description

To demonstrate self-managing of COTS component-based approach, a distributed factory automation system is considered. In a distributed factory automation system (Gomaa 2000), manufacturing workstations are physically laid out in an assembly line. Parts are moved between workstations on a conveyor belt. A part is processed at each workstation in sequence and variations on a given product can be handled because the workstations are programmable. Each workstation has an assembly robot for assembling the product, and a pick-and-place robot for picking parts off and placing them on the conveyor. Each robot is equipped with sensors and actuators. The sensors are used for monitoring operating conditions such as detecting part arrival, while actuators are used to switch automation equipment on and off such as the conveyor. Only Receiving Workstation Controller (the front) and Shipping Workstation Controller (the last) have a pick-and-place robot. All other workstations, referred to as Line Workstation Controllers, have the assembly robots in addition to pick-and-place robots.

The manufacturing steps required to manufacture a given part in the factory, from raw material to finished product, are defined in a process plan. The process plan defines

the part type and the sequence of manufacturing operations. Each operation is carried out at a workstation. Process engineers create process plans and their constituent operations.

The processing of new parts in the factory is initiated by the creation of a part order by a human production manager. The part order defines the quantity of parts required for a given part type.

9.3. Software Architecture

In the software architecture, each Line Workstation Controller COTS Component is allocated to a manufacturing workstation, while the Receiving Workstation Controller COTS Component and the Shipping Workstation Controller COTS Component are allocated to a receiving workstation and to a shipping workstation respectively. The Production Manager COTS Component and the Process Engineer COTS Component are allocated to two separate nodes for performance reasons.

At the initialization of the system, the Receiving Workstation Controller COTS Component sends a part request message to the Production Manager COTS Component to let it know that the receiving workstation controller is ready to operate. The Line Workstation Controller COTS Components and the Shipping Workstation Controller COTS Component also send to the predecessor workstation (Receiving Workstation Controller COTS Component or Line Workstation Controller COTS Component) a part request message to let it know that the workstation is ready to operate.

The production manager starts the manufacturing of a process plan introducing the process plan identification in the Production Manager COTS Component. The Production Manager COTS Component creates the part orders corresponding to the

process plan received, and starts the production of the first part order that has not been finished yet. If the receiving workstation controller is ready to operate, the Production Manager COTS Component sends to the Receiving Workstation Controller COTS Component a start part message.

The Receiving Workstation Controller COTS Component is connected to one external system: the Pick-and-Place Robot, as shown in Figure 9.1. Upon the receiving of start part message, the Receiving Workstation Controller COTS Component sends a pick part message to the Pick-and-Place Robot to pick and place the raw material on the conveyor belt. Then, the Receiving Workstation Controller COTS Component sends a part coming message to the successor workstation controller (in this case, the Line 1 Workstation Controller COTS Component) if the workstation is ready to operate.

Figure 9.1: Receiving Workstation Controller COTS Component Software Architecture

The Line Workstation Controller COTS Component is connected to two external systems: the Pick-and-Place Robot and the Assembly Robot, as shown in Figure 9.2. Upon the receiving of the part coming message from the predecessor workstation controller (Receiving Workstation Controller COTS Component or predecessor Line Workstation Controller COTS Component), the Line Workstation Controller COTS Component asks the Production Manager COTS Component for the next operation to be performed and the execution time of the operation if it is the first part to be manufactured on the workstation. The Production Manager COTS Component asks to the Process Engineer COTS Component the corresponding operation time. Then, the Line Workstation Controller COTS Component sends a pick and place message to the Pick-and-Place Robot to pick and place the part off the conveyor belt. Then, the Line Workstation Controller COTS Component sends an assembly message to the Assembly Robot to assembly the part. When the part has been assembled, the Line Workstation Controller COTS Component sends an assembled part to the Pick-and-Place Robot to pick and place the assembled part on the conveyor belt. Then, if the successor workstation controller (Line Workstation Controller COTS Component or Shipping Workstation Controller COTS Component) is ready to operate, the Line Workstation Controller COTS Component sends a part coming message, and also sends to the predecessor workstation controller (Receiving Workstation Controller COTS Component or Line Workstation Controller COTS Component) a part request message to indicate that the line workstation controller is ready to operate.

Figure 9.2: Line Workstation Controller COTS Component Software Architecture

The Shipping Workstation Controller COTS Component is connected to one external system: the Pick-And-Place Robot, as shown in Figure 9.3. Upon the receiving of the part coming message from the predecessor workstation controller (last Line Workstation Controller COTS Component), the Shipping Workstation Controller COTS Component sends a pick and place message to the Pick-and-Place Robot to pick and place the finished part off the conveyor. When the part has been placed in storage, the Shipping Workstation Controller COTS Component sends a part request message to the processor workstation controller (last Line Workstation Controller COTS Component) to let it know that the workstation is ready to operate. When the Shipping Workstation Controller COTS Component has received the last part, the Shipping Workstation

Controller sends a part order complete message to the Production Manager COTS

Component to let it know that the part orders has been completed.

Figure 9.3: Shipping Workstation Controller COTS Component Software Architecture

9.4 Implementation of the Distributed Factory Automation System

By consideration of reusability, the COTS component wrapper that plays a key role for self-managing COTS components is designed and implemented by means of both application-independent part and application-dependent part. Application independent part of a COTS component wrapper is common to all COTS components so that it is reused for different COTS component once it is implemented (see chapter 7). Application dependent part of a wrapper contains information specific to a COTS component, which should be updated for a different COTS components. Application-dependent COTS information is described in a COTS component wrapper by means of tables. The

following sections presents the tables used by each one of the COTS component wrappers involved in this case study.

9.4.1 Production Manager COTS Component Wrapper

The Production Manager COTS component wrapper self-manages the Production Manager COTS Component using the application independent algorithms presented in chapter 7 and the application-dependent information as described in the following sections. The application-dependent information has been implemented by means of tables and these tables have been assigned to the Production Manager COTS component wrapper's objects.

9.4.1.1 Production Manager Modified Interface Object

The Production Manager Modified Interface object provides the Production Manager Operation Status Table (Table 9.1), the Production Manager Operation Request Parameter Table (Table 9.2), the Production Manager Operation Return Type Table (Table 9.3), the Production Manager Successor Operation Table (Table 9.4), and the Production Manager Function Status Table (Table 9.5).

Table 9.1: Production Manager Operation Status Table

Operation	Status
createPartOrders	Unblock
startProduction	Unblock
partRequest	Unblock
nextOperationRequest	Unblock
partOrderComplete	Unblock

Table 9.2: Production Manager Operation Request Parameter Table

Operation	Parameter	Type	Order
createPartOrders	Process Plan ID	integer	1
startProduction	Start Production?	string	2
partRequest	Receiving WC Ready?	integer	1
nextOperationRequest	Line WC Number	integer	1
nextOperationRequest	Process Plan ID	integer	3
nextOperationRequest	Part Type ID	integer	4
nextOperationRequest	Last Operation ID	integer	5
partOrderComplete	Process Plan ID	integer	1
partOrderComplete	Part Type ID	integer	3

Table 9.3: Production Manager Operation Return Type Table

Operation	Parameter	Type	Order
createPartOrders	Created?	string	2
startProduction	Process Plan ID	integer	2
startProduction	Part Type ID	integer	3
startProduction	Quantity	integer	4
startProduction	Last Operation ID	integer	5
partRequest	Acknowledge	string	2
nextOperationRequest	Line WC Number	integer	1
nextOperationRequest	Next Operation ID	integer	3
nextOperationRequest	Operation Time	integer	4
partOrderComplete	Acknowledge	string	2

Table 9.4: Production Manager Successor Operation Table

Operation	Successor	
	Operation	Component
createPartOrders	startProduction	Production Manager
startProduction	startPart	Receiving Workstation Controller
partRequest	acknowledge	
nextOperationRequest	response	Line Workstation Controller
partOrderComplete	acknowledge	

Table 9.5: Production Manager Function Status Table

Operation	Status
startProduction	Unblock
startPart	Unblock

9.4.1.2 Production Manager Monitor Object

The Production Manager Monitor Object provides the Production Manager Operation Statechart Table (Table 9.6) and Production Manager Operation Statechart Information Table (Table 9.7).

Table 9.6: Production Manager Operation Statechart Table

Operation	State	Event	Action	Next State
createPartOrders	Idle	createPartOrdersArrived		Calling Create Part Orders
createPartOrders	Calling Create Part Orders	createPartOrdersCalled	Start Timing	Processing Create Part Orders
createPartOrders	Processing Create Part Orders	startProductionArrived	End Timing	Idle
startProduction	Calling Start Production	startProductionCalled	Start Timing	Processing Start Production
startProduction	Processing Start Production	startPartArrived	End Timing	Calling Start Part
startProduction	Calling Start Part	startPartCalled		Calling Start Production
partRequest	Idle	partRequestArrived		Calling Part Request
partRequest	Calling Part Request	partRequestCalled	Start Timing	Processing Part Request
partRequest	Processing Part Request	acknowledgeArrived	End Timing	Idle
nextOperationRequest	Idle	nextOperationRequestArrived		Calling Next Operation Request
nextOperationRequest	Calling Next Operation Request	nextOperationRequestCalled	Start Timing	Processing Next Operation Request
nextOperationRequest	Processing Next Operation Request	responseArrived	End Timing	Calling Response
nextOperationRequest	Calling Response	responseCalled		Idle
partOrderComplete	Idle	partOrderCompleteArrived		Calling Part Order Complete
partOrderComplete	Calling Part Order Complete	partOrderCompleteCalled	Start Timing	Processing Part Order Complete
partOrderComplete	Processing Part Order Complete	acknowledgeArrived	End Timing	Idle

Table 9.7: Production Manager Operation Statechart Information Table

Operation	Current State	Maximum Operation Time	Start Time	End Time
createPartOrders	Idle	8000		
startProduction	Calling Start Production	10000		
partRequest	Idle	7000		
nextOperationRequest	Idle	100000		
partOrderComplete	Idle	10000		

9.4.1.3 Production Manager Wrapper Controller Object

The Production Manager Wrapper Controller object provides the Production Manager Operation Criticality Table (Table 9.8).

Table 9.8: Production Manager Operation Criticality Table

Operation	Criticality
createPartOrders	Critical
startProduction	Critical
partRequest	Critical
nextOperationRequest	Critical
partOrderComplete	Critical

9.4.1.4 Production Manager Reconfiguration Manager Object

The Production Manager Reconfiguration Manager object has been implemented by means of Production Manager Reconfiguration Plan Generator and Production Manager Reconfiguration Plan Executor objects. Both objects have access to the Production Manager Operation Reconfiguration Status Table (Table 9.9) whereas the Production Manager Reconfiguration Plan Generator has access to the Production Manager Reconfiguration Table (Table 9.10).

Table 9.9: Production Manager Operation Reconfiguration Status Table

Operation	Status
createPartOrders	Unblock
startProduction	Unblock
partRequest	Unblock
nextOperationRequest	Unblock
partOrderComplete	Unblock
startPart	Unblock
partPlaced	Unblock
operationRequest	Unblock
partComing	Unblock

Table 9.10: Production Manager Reconfiguration Table

Function	Callee			Caller	
	Function	Function Block	Component	Function	Component
createPartOrders	startPart		Receiving Workstation Controller		
	partPlaced		Receiving Workstation Controller		
startProduction					
partRequest				partPlaced	Line 1 Workstation Controller
nextOperationRequest	operationRequest		Process Engineer	partComing	Line 1 Workstation Controller
				partComing	Line 2 Workstation Controller
				partComing	Line 3 Workstation Controller
partOrderComplete				partPlaced	Shipping Workstation Controller

9.4.1.5 Production Manager Repair Manager Object

The Production Manager Repair Manager object provides the Production Manager Repair Techniques Table (Table 9.11).

Table 9.11: Production Manager Repair Techniques Table

Technique	Status
Re-Install component	Non-Used
Replace component	Non-Used

9.4.2 Process Engineer COTS Component Wrapper

The Process Engineer COTS component wrapper self-manages the Process Engineer COTS Component using the application independent algorithms presented in chapter 7 and the application-dependent information as described in the following sections. The application-dependent information has been implemented by means of tables and these tables have been assigned to the Process Engineer COTS component wrapper's objects.

9.4.2.1 Process Engineer Modified Interface Object

The Process Engineer Modified Interface object provides the Process Engineer Operation Status Table (Table 9.12), the Process Engineer Operation Request Parameter Table (Table 9.13), the Process Engineer Operation Return Type Table (Table 9.14), the Process Engineer Successor Operation Table (Table 9.15), and the Process Engineer Function Status Table (Table 9.16).

Table 9.12: Process Engineer Operation Status Table

Operation	Status
operationRequest	Unblock

Table 9.13: Process Engineer Operation Request Parameter Table

Operation	Parameter	Type	Order
operationRequest	Operation ID	integer	1
operationRequest	PM COTS Component	integer	3

Table 9.14: Process Engineer Operation Return Type Table

Operation	Parameter	Type	Order
operationRequest	Operation Time	integer	1
operationRequest	PM COTS Component	integer	3

Table 9.15: Process Engineer Successor Operation Table

Operation	Successor	
	Operation	Component
operationRequest	acknowledge	Production Manager

Table 9.16: Process Engineer Function Status

Operation	Status
acknowledge	Unblock

9.4.2.2 Process Engineer Monitor Object

The Process Engineer Monitor Object provides the Process Engineer Operation Statechart Table (Table 9.17) and Process Engineer Operation Statechart Information Table (Table 9.18).

Table 9.17: Process Engineer Operation Statechart Table

Operation	State	Event	Action	Next State
operationRequest	Idle	operationRequestArrived		Calling Operation Request
operationRequest	Calling Operation Request	operationRequestCalled	Start Timing	Processing Operation Request
operationRequest	Processing Operation Request	acknowledgeArrived	End Timing	Calling Acknowledge
operationRequest	Calling Acknowledge	acknowledgeCalled		Idle

Table 9.18: Process Engineer Operation Statechart Information Table

Operation	Current State	Maximum Operation Time	Start Time	End Time
operationRequest	Idle	100000		

9.4.2.3 Process Engineer Wrapper Controller Object

The Process Engineer Wrapper Controller object provides the Process Engineer Operation Criticality Table (Table 9.19).

Table 9.19: Process Engineer Operation Criticality Table

Operation	Criticality
operationRequest	Critical

9.4.2.4 Process Engineer Reconfiguration Manager Object

The Process Engineer Reconfiguration Manager object has been implemented by means of Process Engineer Reconfiguration Plan Generator and Process Engineer Reconfiguration Plan Executor objects. Both objects have access to the Process Engineer

Operation Reconfiguration Status Table (Table 9.20) whereas the Process Engineer Reconfiguration Plan Generator has access to the Process Engineer Reconfiguration Table (Table 9.21).

Table 9.20: Process Engineer Operation Reconfiguration Status Table

Operation	Status
operationRequest	Unblock
nextOperationRequest	Unblock

Table 9.21: Process Engineer Reconfiguration Table

Function	Callee			Caller	
	Function	Function Block	Component	Function	Component
operationRequest				nextOperationRequest	Production Manager

9.4.2.5 Process Engineer Repair Manager Object

The Process Engineer Repair Manager object provides the Process Engineer Repair Techniques Table (Table 9.22).

Table 9.22: Process Engineer Repair Techniques Table

Technique	Status
Re-Install component	Non-Used
Replace component	Non-Used

9.4.3 Receiving Workstation Controller COTS Component Wrapper

The Receiving Workstation Controller COTS component wrapper self-manages the Receiving Workstation Controller COTS Component using the application independent algorithms presented in chapter 7 and the application-dependent information as described in the following sections. The application-dependent information has been implemented by means of tables and these tables have been assigned to the Receiving Workstation Controller COTS component wrapper's objects.

9.4.3.1 Receiving Workstation Controller Modified Interface Object

The Receiving Workstation Controller Modified Interface object provides the Receiving Workstation Controller Operation Status Table (Table 9.23), the Receiving Workstation Controller Operation Request Parameter Table (Table 9.24), the Receiving Workstation Controller Operation Return Type Table (Table 9.25), the Receiving Workstation Controller Successor Operation Table (Table 9.26), and the Receiving Workstation Controller Function Status Table (Table 9.27).

Table 9.23: Receiving Workstation Controller Operation Status Table

Operation	Status
startPart	Unblock
partPlaced	Unblock
partRequest	Unblock

Table 9.24: Receiving Workstation Controller Operation Request Parameter Table

Operation	Parameter	Type	Order
startPart	Process Plan ID	integer	1
startPart	Part Type ID	integer	3
startPart	Quantity	integer	4
startPart	Last Operation ID	integer	5
partPlaced	Part was Placed?	string	2
partRequest	Line 1 WC Ready?	string	2

Table 9.25: Receiving Workstation Controller Operation Return Type Table

Operation	Parameter	Type	Order
startPart	Pick Part?	string	2
partPlaced	Process Plan ID	integer	1
partPlaced	Part Type ID	integer	3
partPlaced	Quantity	integer	4
partPlaced	Last Operation ID	integer	5
partRequest	Acknowledge	string	2

Table 9.26: Receiving Workstation Controller Successor Operation Table

Operation	Successor	
	Operation	Component
startPart	pickPart	Pick-and-Place Robot External System
partPlaced	partComing	Line 1 Workstation Controller
partRequest	acknowledge	

Table 9.27: Receiving Workstation Controller Function Status

Operation	Status
pickPart	Unblock
partComing	Unblock

9.4.3.2 Receiving Workstation Controller Monitor Object

The Receiving Workstation Controller Monitor Object provides the Receiving Workstation Controller Operation Statechart Table (Table 9.28) and Receiving Workstation Controller Operation Statechart Information Table (Table 9.29).

Table 9.28: Receiving Workstation Controller Operation Statechart Table

Operation	State	Event	Action	Next State
startPart	Idle	startPartArrived		Calling Start Part
startPart	Calling Start Part	startPartCalled	Start Timing	Processing Start Part
startPart	Processing Start Part	pickPartArrived	End Timing	Calling Pick Part
startPart	Calling Pick Part	pickPartCalled		Idle
partPlaced	Idle	partPlacedArrived		Calling Part Placed
partPlaced	Calling Part Placed	partPlacedCalled	Start Timing	Processing Part Placed
partPlaced	Processing Part Placed	partComingArrived	End Timing	Calling Part Coming
partPlaced	Calling Part Coming	partComingCalled		Idle
partRequest	Idle	partRequestArrived		Calling Part Request
partRequest	Calling Part Request	partRequestCalled	Start Timing	Processing Part Request
partRequest	Processing Part Request	acknowledgeArrvied	End Timing	Idle

Table 9.29: Receiving Workstation Controller Operation Statechart Information Table

Operation	Current State	Maximum Operation Time	Start Time	End Time
startPart	Idle	10000		
partPlaced	Idle	10000		
partRequest	Idle	10000		

9.4.3.3 Receiving Workstation Controller Wrapper Controller Object

The Receiving Workstation Controller Wrapper Controller object provides the Receiving Workstation Controller Operation Criticality Table (Table 9.30).

Table 9.30: Receiving Workstation Controller Operation Criticality Table

Operation	Criticality
startPart	Critical
partPlaced	Critical
partRequest	Critical

9.4.3.4 Receiving Workstation Controller Reconfiguration Manager Object

The Receiving Workstation Controller Reconfiguration Manager object has been implemented by means of Receiving Workstation Controller Reconfiguration Plan Generator and Receiving Workstation Controller Reconfiguration Plan Executor objects. Both objects have access to the Receiving Workstation Controller Operation Reconfiguration Status Table (Table 9.31) whereas the Receiving Workstation Controller Reconfiguration Plan Generator has access to the Receiving Workstation Controller Reconfiguration Table (Table 9.32).

Table 9.31: Receiving Workstation Controller Operation Reconfiguration Status Table

Operation	Status
startPart	Unblock
partPlaced	Unblock
partRequest	Unblock
partComing	Unblock
partAssembled	Unblock
partAssembledPlaced	Unblock
createPartOrders	Unblock

Table 9.32: Receiving Workstation Controller Reconfiguration Table

Function	Callee			Caller	
	Function	Function Block	Component	Function	Component
startPart				createPartOrders	Production Manager
partPlaced	partRequest		Production Manager	createPartOrders	Production Manager
	partComing		Line 1 Workstation Controller		
	partRequest		Line 1 Workstation Controller		
	partPlaced		Line 1 Workstation Controller		
	partAssembled		Line 1 Workstation Controller		
	partAssembledPlaced		Line 1 Workstation Controller		
partRequest				partAssembledPlaced	Line 1 Workstation Controller

9.4.3.5 Receiving Workstation Controller Repair Manager Object

The Receiving Workstation Controller Repair Manager object provides the Receiving Workstation Controller Repair Techniques Table (Table 9.33).

Table 9.33: Receiving Workstation Controller Repair Techniques Table

Technique	Status
Re-Install component	Non-Used
Replace component	Non-Used

9.4.4 Line Workstation Controller COTS Component Wrapper

The Line Workstation Controller COTS component wrapper self-manages the Line Workstation Controller COTS Component using the application independent algorithms presented in chapter 7 and the application-dependent information as described in the following sections. The application-dependent information has been implemented

by means of tables and these tables have been assigned to the Line Workstation Controller COTS component wrapper's objects.

9.4.4.1 Line Workstation Controller Modified Interface Object

The Line Workstation Controller Modified Interface object provides the Line Workstation Controller Operation Status Table (Table 9.34), the Line Workstation Controller Operation Request Parameter Table (Table 9.35), the Line Workstation Controller Operation Return Type Table (Table 9.36), the Line Workstation Controller Successor Operation Table (Table 9.37), and the Line Workstation Controller Function Status Table (Table 9.38).

Table 9.34: Line Workstation Controller Operation Status Table

Operation	Status
partComing	Unblock
partPlaced	Unblock
partAssembled	Unblock
partAssembledPlaced	Unblock
partRequest	Unblock

Table 9.35: Line Workstation Controller Operation Request Parameter Table

Operation	Parameter	Type	Order
partComing	Process Plan ID	integer	1
partComing	Part Type ID	integer	3
partComing	Quantity	integer	4
partComing	Last Operation ID	integer	5
partPlaced	Part was Placed?	string	2
partAssembled	Part was Assembled?	string	2
partAssembledPlaced	Part Assembled was Placed?	string	2
partRequest	Successor WC Ready?	string	2

Table 9.36: Line Workstation Controller Operation Return Type Table

Operation	Parameter	Type	Order
partComing	Pick Part?	string	2
partPlaced	Operation ID	integer	1
partPlaced	Operation Time	integer	3
partAssembled	acknowledge	string	2
partAssembledPlaced	Process Plan ID	integer	1
PartAssembledPlaced	Part Type ID	integer	3
partAssembledPlaced	Quantity	integer	4
partAssembledPlaced	Last Operation ID	integer	5
partRequest	Acknowledge	string	2

Table 9.37: Line Workstation Controller Successor Operation Table

Operation	Successor	
	Operation	Component
partComing	pickPart	Pick-and-Place Robot External System
partPlaced	assemblePart	Assembly Robot External System
partAssembled	acknowledge	
partAssembledPlaced	partComing	Successor Workstation Controller
partRequest	acknowledge	

Table 9.38: Line Workstation Controller Function Status

Operation	Status
pickPart	Unblock
assemblePart	Unblock
partComing	Unblock

9.4.4.2 Line Workstation Controller Monitor Object

The Line Workstation Controller Monitor Object provides the Line Workstation Controller Operation Statechart Table (Table 9.39) and Line Workstation Controller Operation Statechart Information Table (Table 9.40).

Table 9.39: Line Workstation Controller Operation Statechart Table

Operation	State	Event	Action	Next State
partRequest	Idle	partRequestArrived		Calling Part Request
partRequest	Calling Part Request	partRequestCalled	Start Timing	Processing Part Request
partRequest	Processing Part Request	acknowledgeArrived	End Timing	Idle
partComing	Idle	partComingArrived		Calling Part Coming
partComing	Calling Part Coming	partComingCalled	Start Timing	Processing Part Coming
partComing	Processing Part Coming	pickPartArrived	End Timing	Calling Pick Part
partComing	Calling Pick Part	pickPartCalled		Idle
partPlaced	Idle	partPlacedArrived		Calling Part Placed
partPlaced	Calling Part Placed	partPlacedCalled	Start Timing	Processing Part Placed
partPlaced	Processing Part Placed	assemblePartArrived	End Timing	Calling Assemble Part
partPlaced	Calling Assemble Part	assemblePartCalled		Idle
partAssembled	Idle	partAssembledArrived		Calling Part Assembled
partAssembled	Calling Part Assembled	partAssembledCalled	Start Timing	Processing Part Assembled
partAssembled	Processing Part Assembled	acknowledgeArrived	End Timing	Idle
partAssembledPlaced	Idle	partAssembledPlacedArrived		Calling Part Assembled Placed
partAssembledPlaced	Calling Part Assembled Placed	partAssembledPlacedCalled	Start Timing	Processing Part Assembled Placed
partAssembledPlaced	Processing Part Assembled Placed	partComingArrived	End Timing	Calling Part Coming
partAssembledPlaced	Calling Part Coming	partComingCalled		Idle

Table 9.40: Line Workstation Controller Operation Statechart Information Table

Operation	Current State	Maximum Operation Time	Start Time	End Time
partComing	Idle	100000		
partPlaced	Idle	10000		
partAssembled	Idle	10000		
partAssembledPlaced	Idle	20000		
partRequest	Idle	15000		

9.4.4.3 Line Workstation Controller Wrapper Controller Object

The Line Workstation Controller Wrapper Controller object provides the Line Workstation Controller Operation Criticality Table (Table 9.41).

Table 9.41: Line Workstation Controller Operation Criticality Table

Operation	Criticality
partComing	Critical
partPlaced	Critical
partAssembled	Critical
partAssembledPlaced	Critical
partRequest	Critical

9.4.4.4 Line Workstation Controller Reconfiguration Manager Object

The Line Workstation Controller Reconfiguration Manager object has been implemented by means of Line Workstation Controller Reconfiguration Plan Generator and Line Workstation Controller Reconfiguration Plan Executor objects. Both objects have access to the Line Workstation Controller Operation Reconfiguration Status Table (Table 9.42) whereas the Line Workstation Controller Reconfiguration Plan Generator has access to the Line Workstation Controller Reconfiguration Table (Table 9.43).

Table 9.42: Line Workstation Controller Operation Reconfiguration Status Table

Operation	Status
partComing	Unblock
partPlaced	Unblock
partAssembled	Unblock
partAssembledPlaced	Unblock
partRequest	Unblock
nextOperationRequest	Unblock

Table 9.43: Line Workstation Controller Reconfiguration Table

Function	Callee			Caller	
	Function	Function Block	Component	Function	Component
partComing	nextOperationRequest		Production Manager	partPlaced	Predecessor Workstation Controller
partPlaced				partPlaced	Predecessor Workstation Controller
partAssembled				partPlaced	Predecessor Workstation Controller
partAssembledPlaced	partRequest		Predecessor Workstation Controller	partPlaced	Predecessor Workstation Controller
	partComing		Successor Workstation Controller		
partRequest				partAssembledPlaced	Successor Workstation Controller

9.4.4.5 Line Workstation Controller Repair Manager Object

The Line Workstation Controller Repair Manager object provides the Line Workstation Controller Repair Techniques Table (Table 9.44).

Table 9.44: Line Workstation Controller Repair Techniques Table

Technique	Status
Re-Install component	Non-Used
Replace component	Non-Used

9.4.5 Shipping Workstation Controller COTS Component Wrapper

The Shipping Workstation Controller COTS component wrapper self-manages the Shipping Workstation Controller COTS Component using the application independent algorithms presented in chapter 7 and the application-dependent information as described in the following sections. The application-dependent information has been implemented by means of tables and these tables have been assigned to the Shipping Workstation Controller COTS component wrapper's objects.

9.4.5.1 Shipping Workstation Controller Modified Interface Object

The Shipping Workstation Controller Modified Interface object provides the Shipping Workstation Controller Operation Status Table (Table 9.45), the Shipping Workstation Controller Operation Request Parameter Table (Table 9.46), the Shipping Workstation Controller Operation Return Type Table (Table 9.47), the Shipping Workstation Controller Successor Operation Table (Table 9.48), and the Shipping Workstation Controller Function Status Table (Table 9.49).

Table 9.45: Shipping Workstation Controller Operation Status Table

Operation	Status
partComing	Unblock
partPlaced	Unblock

Table 9.46: Shipping Workstation Controller Operation Request Parameter Table

Operation	Parameter	Type	Order
partComing	Process Plan ID	integer	1
partComing	Part Type ID	integer	3
partComing	Quantity	integer	4
partComing	Last Operation ID	integer	5
partPlaced	Part was Placed?	string	2

Table 9.47: Shipping Workstation Controller Operation Return Type Table

Operation	Parameter	Type	Order
partComing	Pick Part?	string	2
partPlaced	Acknowledge	string	2

Table 9.48: Shipping Workstation Controller Successor Operation Table

Operation	Successor	
	Operation	Component
partComing	pickPart	Pick-and-Place Robot External System
partPlaced	acknowledge	

Table 9.49: Shipping Workstation Controller Function Status

Operation	Status
pickPart	Unblock

9.4.5.2 Shipping Workstation Controller Monitor Object

The Shipping Workstation Controller Monitor Object provides the Shipping Workstation Controller Operation Statechart Table (Table 9.50) and Shipping Workstation Controller Operation Statechart Information Table (Table 9.51).

Table 9.50: Shipping Workstation Controller Operation Statechart Table

Operation	State	Event	Action	Next State
partComing	Idle	partComingArrived		Calling Part Coming
partComing	Calling Part Coming	partComingCalled	Start Timing	Processing Part Coming
partComing	Processing Part Coming	pickPartArrived	End Timing	Calling Pick Part
partComing	Calling Pick Part	pickPartCalled		Idle
partPlaced	Idle	partPlacedArrived		Calling Part Placed
partPlaced	Calling Part Placed	partPlacedCalled	Start Timing	Processing Part Placed
partPlaced	Processing Part Placed	acknowledgeArrived	End Timing	Idle

Table 9.51: Shipping Workstation Controller Operation Statechart Information Table

Operation	Current State	Maximum Operation Time	Start Time	End Time
partComing	Idle	10000		
partPlaced	Idle	10000		

9.4.5.3 Shipping Workstation Controller Wrapper Controller Object

The Shipping Workstation Controller Wrapper Controller object provides the Shipping Workstation Controller Operation Criticality Table (Table 9.52).

Table 9.52: Shipping Workstation Controller Operation Criticality Table

Operation	Criticality
partComing	Critical
partPlaced	Critical

9.4.5.4 Shipping Workstation Controller Reconfiguration Manager Object

The Shipping Workstation Controller Reconfiguration Manager object has been implemented by means of Shipping Workstation Controller Reconfiguration Plan Generator and Shipping Workstation Controller Reconfiguration Plan Executor objects. Both objects have access to the Shipping Workstation Controller Operation Reconfiguration Status Table (Table 9.53) whereas the Shipping Workstation Controller Reconfiguration Plan Generator has access to the Shipping Workstation Controller Reconfiguration Table (Table 9.54).

Table 9.53: Shipping Workstation Controller Operation Reconfiguration Status Table

Operation	Status
partComing	Unblock
partPlaced	Unblock
partAssembledPlaced	Unblock
partRequest	Unblock
partOrderComplete	Unblock

Table 9.54: Shipping Workstation Controller Reconfiguration Table

Function	Callee			Caller	
	Function	Function Block	Component	Function	Component
partComing				partAssembledPlaced	Predecessor Workstation Controller
partPlaced	partRequest		Predecessor Workstation Controller	partAssembledPlaced	Predecessor Workstation Controller
	partOrderComplete		Production Manager		Predecessor Workstation Controller

9.4.5.5 Shipping Workstation Controller Repair Manager Object

The Shipping Workstation Controller Repair Manager object provides the Shipping Workstation Controller Repair Techniques Table (Table 9.55).

Table 9.55: Shipping Workstation Controller Repair Techniques Table

Technique	Status
Re-Install component	Non-Used
Replace component	Non-Used

9.5 Validation of the Distributed Factory Automation System

This section presents the validation of the mechanisms provided by a *COTS component wrapper* for self-managing COTS components. The application-independent algorithms of a *COTS component wrapper* are validated to determine if the algorithms can detect, reconfigure, and recover anomalies of COTS components correctly.

9.5.1 Hardware and Software Specifications

These test cases used to validate the Detection Mechanism (chapter 4), the Reconfiguration Mechanism (chapter 5) and the Repair Mechanism (chapter 6) were executed by a computer with the following hardware and software specifications:

Hardware:

- Processor: Intel® Core™2 CPU – T5500
- Speed: 1.66 GHz – 1.67 GHz.
- Memory (RAM): 1.00 GB
- System type: 32-bit Operating System

Software:

- Operative System: Windows Vista™ Home Premium with Service Pack 1
- Development Environment: Eclipse Platform Version 3.2.0
- Java™ SE Runtime Environment: 1.6.0-rc

9.5.2 Validation of the Detection Mechanism's Algorithms

The Detection Mechanism (chapter 4) was also validated using the second case-study, the Distributed Factory Automation System (Gomaa 2000). The validation of the Detection Mechanism involves the procedure of the testing, the related objects involved in the *COTS component wrapper*, the application-independent algorithms provided by each object, the test cases performed, and the result of those test cases.

9.5.2.1 Testing Procedure for Detection Mechanism

- 1) Execute the *system* three times to obtain the execution times for each function in each COTS component.
- 2) Establish the *maximum* execution time for each function in the Operation Statechart Information Table as the average of the three execution times plus a 50%.
- 3) Check if the *COTS Modified Interface* object extracts correctly the function name from the parameters received.
- 4) Verify if the *COTS Modified Interface* object extracts and shows to the console the function's arguments using the Operation Request Parameter Table.

- 5) Validate if the *COTS Modified Interface* object checks the status of each function arrived in the Operation Status Table and allows the function to be invoked from the COTS component only when the status is 'unblock'. Otherwise, the *COTS Modified Interface* object only displays a message indicating that the function is blocked.
- 6) Check if the *COTS Modified Interface* object sends the functionName + 'Arrived' event to the *COTS Monitor* object, and if the function's statechart transits to the corresponding state using the Operation Statechart Table and the Operation Statechart Information Table.
- 7) Verify if the *COTS Modified Interface* object calls the function from the COTS component with the corresponding arguments.
- 8) Check if the *COTS Modified Interface* object sends the functionName + 'Called' event to the *COTS Monitor* object, and if the function's statechart transits to the corresponding state using the Operation Statechart Table and the Operation Statechart Information Table.
- 9) Validate if the *COTS Monitor* object starts timing the function based on the event received using the maximum execution time established in the Operation Statechart Information Table.
- 10) Verify if the *COTS Modified Interface* object receives the replay from the COTS component extracting and showing its arguments using the Operation Return Type Table.
- 11) Check if the *COTS Modified Interface* object identifies correctly the successor function in the Successor Operation Table.

- 12) Validate if the *COTS Modified Interface* object sends the *successorFunction* + ‘Arrived’ event to the *COTS Monitor* object, and if the execution time for that function is less or equal to the maximum execution time recorded in the Statechart Operation Table. If this condition is satisfied, check if the function’s statechart transits to the corresponding state using the Operation Statechart Table and the Operation Statechart Information Table. If this condition is not satisfied, check if the *COTS Monitor* object detects the error and communicates the function name to the *Wrapper Controller* object.
- 13) If the successor function is provided by a COTS component, verify the following: if the *COTS Modified Interface* object calls the successor function, if the *successorFunction* + ‘Called’ event is sent to the *COTS Monitor* object by the *COTS Modified Interface* object, and if the function’s statechart transits to the Idle state.
- 14) If the successor function is just an acknowledgment, verify that the function’s statechart transits to the Idle state based on the *successorFunction* + ‘Arrived’ event sent by the *COTS Modified Interface* object to the *COTS Monitor* object in step 12.
- 15) If the *COTS Modified Interface* object does not receive the replay from the COTS component due to an error in the COTS component (step 10), and the maximum execution time has expired, check if the *COTS Monitor* object detects the error and sends the function name to the *Wrapper Controller* object.

9.5.2.2 Related Objects in COTS Component Wrapper

The Detection Mechanism is supported by the following COTS component wrapper objects: COTS Modified Interface, COTS Monitor, and Wrapper Controller objects.

9.5.2.3 Operations Provided by each Object in the COTS Component Wrapper

Table 9.56 shows the application-independent algorithms provided by each object of the COTS component wrapper involved in the Detection Mechanism.

Table 9.56: Application-Independent Algorithms for Detection Mechanism

Algorithm	COTS component wrapper object
OperationRequest(in parameters)	COTS Modified Interface object
ModifiedInterfaceNotification(in operation, event)	COTS Monitor object
StartTime(in operation, time)	COTS Monitor object
MonitorTimeNotification(in operation)	COTS Monitor object
MalfunctionOperation(in operation)	Wrapper Controller object

9.5.2.4 Testing

To validate the Detection Mechanism in the *Production Manager COTS component*, ten test cases were used, two for each function provided by the Production Manager COTS component: Normal case and Fault or error case.

- 1) **createPartOrders** function in Production Manager COTS component
 - a. Normal case: the function performed its activity normally.
 - b. Fault or error case: a fault was inserted in the Production Manager COTS component preventing the Production Manager COTS

component to reply the successful finalization of the function. In the ProductionManager.java class, the lines 131, and 134 to 138 were eliminated.

2) **startProduction** function in Production Manager COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Production Manager COTS component preventing the Production Manager COTS component to reply the process plan ID, the part type ID, the quantity to be manufactured, and the last operation ID to the Receiving Workstation Controller COTS component wrapper. In the ProductionManager.java class, the lines 162 to 168, 192 to 194, and 209 were eliminated.

3) **partRequest** function in Production Manager COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Production Manager COTS component preventing the Production Manager COTS component to reply a successful execution of the function. In the ProductionManager.java class, the line 228 was eliminated.

4) **nextOperationRequest** function in Production Manager COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Production Manager COTS component preventing the Production Manager COTS component to reply the next operation ID and the execution time of the

operation to the corresponding Line Workstation Controller COTS component. In the `ProductionManager.java` class, the lines 241 to 283 were eliminated.

5) **partOrderComplete** function in Production Manager COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Production Manager COTS component preventing the Production Manager COTS component to reply a successful execution of the function. In the `ProductionManager.java` class, the lines 293 to 323 were eliminated.

The validation of the Detection Mechanism in the Process Engineer COTS component, the Receiving Workstation Controller COTS component, the Line Workstation Controller COTS components, the Shipping Workstation Controller COTS component and the test results are depicted in appendix C.

9.5.2.5 Test Result

The ten tests performed on the *Production Manager COTS component* provide the following results:

1) **createPartOrders** function in Production Manager COTS component

- a. Normal case: the function performed its activity successfully and the COTS Modified Interface object called the corresponding `startProduction` function from the Production Manager COTS component wrapper.

- b. Fault or error case: the COTS Monitor object detected the error in the createPartOrders function, and communicated the malfunction to the Wrapper Controller object who initiated the reconfiguration process.
- 2) **startProduction** function in Production Manager COTS Component
 - a. Normal case: the function performed its activity successfully and the COTS Modified Interface object called the corresponding startPart function from the Receiving Workstation Controller COTS component wrapper.
 - b. Fault or error case: the COTS Monitor object detected the error in the startProduction function, and communicated the malfunction to the Wrapper Controller object who initiated the reconfiguration process.
- 3) **partRequest** function in Production Manager COTS Component
 - a. Normal case: the function performed its activity successfully and the COTS Modified Interface object received acknowledge of successful execution of the function.
 - b. Fault or error case: the COTS Monitor object detected the error in the partRequest function, and communicated the malfunction to the Wrapper Controller object who initiated the reconfiguration process.
- 4) **nextOperationRequest** function in Production Manager COTS Component
 - a. Normal case: the function performed its activity successfully and the COTS Modified Interface object sent the next operation ID and the operation execution time to the corresponding Line Workstation Controller COTS component.

- b. Fault or error case: the COTS Monitor object detected the error in the `nextOperationRequest` function, and communicated the malfunction to the Wrapper Controller object who initiated the reconfiguration process.

5) **partOrderComplete** function in Production Manager COTS Component

- a. Normal case: the function performed its activity successfully and updated the process plan information.
- b. Fault or error case: the COTS Monitor object detected the error in the `partOrderComplete` function, and communicated the malfunction to the Wrapper Controller object who initiated the reconfiguration process.

9.5.3 Validation of the Reconfiguration Mechanism's Algorithms

The Reconfiguration Mechanism (chapter 5) was also validated using the second case-study, the Distributed Factory Automation System (Gomaa 2000). The validation of the Reconfiguration Mechanism involves the procedure of the testing, the related objects involved in the *COTS component wrapper*, the application-independent algorithms provided by each object, the test cases performed, and the result of those test cases.

9.5.3.1 Testing Procedure for Reconfiguration Mechanism

- 1) Check if the *Wrapper Controller* object receives the malfunction function from the *COTS Monitor* object and identifies the criticality of the malfunction function successfully using the Operation Criticality Table.

- 2) Validate if the *Wrapper Controller* object sends the malfunction function with its corresponding action to the *Reconfiguration Plan Generator* object.
- 3) Verify if the *Reconfiguration Plan Generator* object generates the Reconfiguration Plan for the malfunction function using its Operation Status Table, Operation Caller Table, and Operation Callee Table.
- 4) Check if the *Reconfiguration Plan Generator* object sends the Reconfiguration Plan to the *Reconfiguration Plan Executor* object.
- 5) Validate if the *Reconfiguration Plan Executor* object executes each one of the lines established in the Reconfiguration Plan.
- 6) Verify if the *Reconfiguration Plan Executor* object updates the status of each function specified in the Reconfiguration Plan in the Operation Status Table.
- 7) In the case that a function is provided by the COTS component being wrapped, check if the *Reconfiguration Plan Executor* object sends the message to the *COTS Modified Interface* object to perform the action specified in the Reconfiguration Plan. In addition, validate if the *COTS Modified Interface* object notifies to the *Reconfiguration Plan Executor* object when the action has been updated in its Operation Status Table.
- 8) In the case that a function have non dependent functions in other COTS components, checks if the *Reconfiguration Plan Executor* object shows the action to the console.
- 9) In the case that a function is provided by other COTS components, verify if the *Reconfiguration Plan Executor* object sends the message to the *Wrapper*

Controller object to reconfigure other dependents COTS components with the action specified in the Reconfiguration Plan.

- 10) In the case of the reconfiguration after a repair has been performed, check if the *Reconfiguration Plan Executor* object sends the message to the *COTS Monitor* object to reinitialize the statechart of the malfunction operation. In addition, validate if the *COTS Monitor* object notified to the *Reconfiguration Plan Executor* object when the statechart has been reinitialized.
- 11) When the Reconfiguration Plan has been executed, verify if the *Reconfiguration Plan Executor* object notifies the finalization of the reconfiguration process to the *Wrapper Controller* object.
- 12) Check if the *Wrapper Controller* object initiates the repair process when is adequate (case before repair) or shows to the console that the COTS component is in full service again (case after repair).
- 13) Validate if the *Wrapper Controller* object receives the malfunction function from the *Wrapper Controller* object in other COTS components, and performs the reconfiguration process in the COTS component being wrapped.

9.5.3.2 Related Objects in COTS Component Wrapper

The Reconfiguration Mechanism is supported by the following COTS component wrapper objects: Wrapper Controller, Reconfiguration Plan Generator, Reconfiguration Plan Executor, COTS Modified Interface, and COTS Monitor objects.

9.5.3.3 Operations Provided by each Object in the COTS Component Wrapper

Table 9.57 shows the application-independent algorithms provided by each object of the COTS component wrapper involved in the Reconfiguration Mechanism.

Table 9.57: Application-Independent Algorithms for Reconfiguration Mechanism

Algorithm	COTS component wrapper object
MalfunctionOperation(in operation)	Wrapper Controller object
GenerateReconfigurationPlan(in operation, object, action)	Reconfiguration Plan Generator object
ExecuteReconfigurationPlan(in reconfiguration plan)	Reconfiguration Plan Executor object
UpdateOperationStatus(in operation, action)	COTS Modified Interface object
ModifiedInterfaceNotification(in notification)	Reconfiguration Plan Executor object
ReinitializeStatechart(in operation)	COTS Monitor object
MonitorNotification(in notification)	Reconfiguration Plan Executor object
ReconfigurationExecutorNotification(in operation, object, action, notification)	Wrapper Controller object
MalfunctionOperationOtherComponents(in operation, object, action)	Wrapper Controller object

9.5.3.4 Testing

To validate the Reconfiguration Mechanism in the *Production Manager COTS component*, five test cases were used, one for each function provided by the Production Manager COTS component. Each test case contains the following information:

- Criticality of the function.
- Reconfiguration plan against the anomalous function. The reconfiguration plan contains information such as function name, object that provides that function, and the action to be performed.

In the case that an anomalous function contains dependent functions in other COTS components, the reconfiguration plans of those other dependent COTS component are depicted as well.

1) **createPartOrders** function in Production Manager COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Production Manager COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Table 9.58 shows the Production Manager Reconfiguration Plan against the anomalous createPartOrders function.

Table 9.58: Production Manager Reconfiguration Plan for Create Part Orders Function

Line No.	Operation	Object	Action
1	createPartOrders		
2	allOperations	Production Manager	Repair Block
3	createPartOrders		Repair Block
4	createPartOrders	Production Manager	Notify Console
5	startProduction		Repair Block
6	startProduction	Production Manager	Notify Console
7	partRequest		Repair Block
8	partRequest	Receiving Workstation Controller	Repair Block
9	partPlaced		Dependency Block
10	nextOperationRequest		Repair Block
11	nextOperationRequest	Line 1 Workstation Controller	Repair Block
12	partComing		Dependency Block
13	nextOperationRequest	Line 2 Workstation Controller	Repair Block
14	partComing		Dependency Block
15	nextOperationRequest	Line 3 Workstation Controller	Repair Block
16	partComing		Dependency Block
17	partOrderComplete		Repair Block
18	partOrderComplete	Shipping Workstation Controller	Repair Block
19	partPlaced		Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1*: This is the name of the function that is malfunction, so the system can identify the reconfiguration plan.
 - *Line 2*: This function is critical; therefore, all the functions in Production Manager COTS component must be blocked as 'Repair Block'. This line is used to communicate only once to the COTS Modified Interface object to repair block all the operations provided by the Production Manager COTS component.
 - *Lines 3, 5, 7, 9, 10, 12, 14, 16, 17, and 19*: These lines are used to change the status of the functions in the Production Manager Operation Reconfiguration Status Table in the Reconfiguration Manager object.
 - *Lines 4 and 6*: These functions have no dependent operations in other components and because they are critical, a message should be display to the console.
 - *Lines 8, 11, 13, 15, and 18*: These functions have dependent operations in other COTS components; therefore, those functions have to be blocked as 'Repair Block'. These lines are used for the Reconfiguration Plan Executor object to send the malfunction notifications to the other Wrapper Controller objects.
- c. Reconfiguration in the *Receiving Workstation Controller COTS Component*: The Receiving Workstation Controller Wrapper Controller receives one function that is malfunction. The reconfiguration plan is as follows:

- **partRequest** function: Table 9.64 shows the Receiving Workstation Controller Reconfiguration Plan for the Part Request malfunction function.

Table 9.59: Receiving Workstation Controller Reconfiguration Plan for Part Request Function

Line No.	Operation	Object	Action
1	partRequest		
2	partRequest		Repair Block
3	partPlaced		Dependency Block
4	partPlaced	Receiving Workstation Controller	Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1*: This is the function name that is malfunction, so the system can identify the reconfiguration plan.
- *Lines 2 and 3*: These lines are used to change the status of the functions in the Receiving Workstation Controller Operation Reconfiguration Status Table in the Reconfiguration Manager object.
- *Line 4*: This line is used to communicate to the COTS Modified Interface to ‘Dependency Block’ the function.

d. Reconfiguration in the *Line Workstation Controller COTS*

Components: The Line Workstation Controller Wrapper Controllers receive one function that is malfunction. The reconfiguration plan is as follows:

- **nextOperationRequest** function: Table 9.60 shows the Line Workstation Controller Reconfiguration Plan for the Next Operation Request malfunction function.

Table 9.60: Line Workstation Controller Reconfiguration Plan for Next Operation Request Function

Line No.	Operation	Object	Action
1	nextOperationRequest		
2	nextOperationRequest		Repair Block
3	partComing		Dependency Block
4	partComing	Line Workstation Controller	Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1*: This is the function name that is malfunction, so the system can identify the reconfiguration plan.
 - *Lines 2 and 3*: These lines are used to change the status of the functions in the Line Workstation Controller Operation Reconfiguration Status Table in the Reconfiguration Manager object.
 - *Line 4*: This line is used to communicate to the COTS Modified Interface to ‘Dependency Block’ the function.
- e. Reconfiguration in the *Shipping Workstation Controller COTS Component*: The Shipping Workstation Controller Wrapper Controller receives one function that is malfunction. The reconfiguration plan is as follows:

- **partOrderComplete** function: Table 9.61 shows the Shipping Workstation Controller Reconfiguration Plan for the Part Order Complete malfunction function.

Table 9.61: Shipping Workstation Controller Reconfiguration Plan for Part Order Complete Function

Line No.	Operation	Object	Action
1	partOrderComplete		
2	partOrderComplete		Malfunction Block
3	partPlaced		Dependency Block
4	partPlaced	Shipping Workstation Controller	Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1*: This is the function name that is malfunction, so the system can identify the reconfiguration plan.
 - *Lines 2 and 3*: These lines are used to change the status of the functions in the Shipping Workstation Controller Operation Reconfiguration Status Table in the Reconfiguration Manager object.
 - *Line 4*: This line is used to communicate to the COTS Modified Interface to ‘Dependency Block’ the function.
- f. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair showed above. The only difference is that the action to be taken is ‘Unblock’ instead of ‘Repair Block’, ‘Malfunction Block’, or ‘Dependency Block’.

2) **startProduction** function in Production Manager COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Production Manager COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for createPartOrders function (Table 9.58). The only difference is that the line 1 shows startProduction instead of createPartOrders.
- c. Reconfiguration in the other COTS components: The Receiving Workstation Controller, Line Workstation Controller and Shipping Workstation Controller Wrapper Controllers receive the same malfunction function specified in createPartOrders function, generating the same reconfiguration plan depicted in Tables 9.59, 9.60, and 9.61.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is 'Unblock' instead of 'Repair Block', 'Malfunction Block', or 'Dependency Block'.

3) **partRequest** function in Production Manager COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Production Manager COTS component needs to be repaired immediately.
 - b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for createPartOrders function (Table 9.58). The only difference is that the line 1 shows partRequest instead of createPartOrders.
 - c. Reconfiguration in the other COTS components: The Receiving Workstation Controller, Line Workstation Controller and Shipping Workstation Controller Wrapper Controllers receive the same malfunction function specified in createPartOrders function, generating the same reconfiguration plan depicted in Tables 9.59, 9.60, and 9.61.
 - d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is 'Unblock' instead of 'Repair Block', 'Malfunction Block', or 'Dependency Block'.
- 4) **nextOperationRequest** function in Production Manager COTS component
- a. Criticality of the function: this function is critical for the system; therefore, the Production Manager COTS component needs to be repaired immediately.

- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for createPartOrders function (Table 9.58). The only difference is that the line 1 shows nextOperationRequest instead of createPartOrders.
- c. Reconfiguration in the other COTS components: The Receiving Workstation Controller, Line Workstation Controller and Shipping Workstation Controller Wrapper Controllers receive the same malfunction function specified in createPartOrders function, generating the same reconfiguration plan depicted in Tables 9.59, 9.60, and 9.61.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is 'Unblock' instead of 'Repair Block', 'Malfunction Block', or 'Dependency Block'.

5) **partOrderComplete** function in Production Manager COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Production Manager COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for createPartOrders function (Table

9.58). The only difference is that the line 1 shows partOrderComplete instead of createPartOrders.

- c. Reconfiguration in the other COTS components: The Receiving Workstation Controller, Line Workstation Controller and Shipping Workstation Controller Wrapper Controllers receive the same malfunction function specified in createPartOrders function, generating the same reconfiguration plan depicted in Tables 9.59, 9.60, and 9.61.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is 'Unblock' instead of 'Repair Block', 'Malfunction Block', or 'Dependency Block'.

The validation of the Reconfiguration Mechanism in the Process Engineer COTS component, the Receiving Workstation Controller COTS component, the Line Workstation Controller COTS components, the Shipping Workstation Controller COTS component and the test results are depicted in appendix C.

9.5.3.5 Test Results

The five tests performed on the *Production Manager COTS component* provide the following results:

- a) Criticality of the operation: the Production Manager Wrapper Controller had successfully identified the criticality of each operation selecting ‘Critical’ for the five test cases.
- b) Reconfiguration Plan before repair: the Production Manager Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test cases 1 to 5 (Tables 9.58, 9.59, 9.60, and 9.61).
- c) Reconfiguration Plan after repair: The Production Manager Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test cases 1 to 5, changing the action to ‘unblock’ (Tables 9.58, 9.59, 9.60, and 9.61).

9.5.4 Validation of the Repair Mechanism’s Algorithms

The Repair Mechanism (chapter 6) was also validated using the second case-study, the Distributed Factory Automation System (Gomaa 2000). The validation of the Repair Mechanism involves the procedure of the testing, the related objects involved in the *COTS component wrapper*, the application-independent algorithms provided by each object, the test cases performed, and the result of those test cases.

9.5.4.1 Testing Procedure for Repair Mechanism

- 1) Check if the Repair Manager receives the notification to start the repair process from the Wrapper Controller.
- 2) Validate if the Repair Manager choose the first repair technique that has not been used yet using the Repair Techniques Table.

- 3) Verify if the Repair Manager performs the repair technique selected.
- 4) Check if the Repair Manager sends the new object generated according to the technique selected to the COTS Modified Interface.
- 5) Validate if the COTS Modified Interface receives the new object and uses this object as a new COTS component.
- 6) Verify if the COTS Modified Interface sends the notification to the Repair Manager indicating that the object has been received.
- 7) Check if the Repair Manager notifies to the Wrapper Controller that the repair process has been executed, and the reconfiguration after repair has to be executed.

9.5.4.2 Related Objects in COTS Component Wrapper

The Repair Mechanism is supported by the following COTS component wrapper objects: Repair Manager, COTS Modified Interface, and Wrapper Controller objects.

9.5.4.3 Operations Provided by each Object in the COTS Component Wrapper

Table 9.62 shows the application-independent algorithms provided by each object of the COTS component wrapper involved in the Repair Mechanism.

Table 9.62: Application-Independent Algorithms for Repair Mechanism

Algorithm	COTS component wrapper object
StartRepairProcess(in component, notification)	Repair Manager object
RepairManagerNotification(in object, notification)	COTS Modified Interface object
RepairExecuted(in notification)	Repair Manager object
RepairManagerNotification(in notification)	Wrapper Controller object

9.5.4.4 Testing

To validate the Repair Mechanism in the *Production Manager COTS component*, two test cases were used, one for each technique available to repair the Production Manager COTS component specified in the Production Manager Repair Techniques Table (Table 9.11).

- 1) Re-install component: The technique re-installs the same component using a copy of the Production Manager COTS component kept in a safe place.
- 2) Replace component: The technique selects a variant version of the Production Manager COTS component as a new Production Manager COTS component.

To validate this test case, the first technique has been marked as ‘used’, so the algorithm will select this second technique.

The validation of the Repair Mechanism in the Process Engineer COTS component, the Receiving Workstation Controller COTS component, the Line Workstation Controller COTS components, the Shipping Workstation Controller COTS component and the test results are depicted in appendix C.

9.5.4.5 Tests Results

The two tests performed on the *Production Manager COTS component* provide the following results:

- 1) Re-install component: The Production Manager Repair Manager object selected and executed this technique to repair the Production Manager COTS

component successfully. The Production Manager COTS component provided full services using a copy of the same Production Manager COTS component.

- 2) Replace component: The Production Manager Repair Manager object selected and executed this second technique successfully. The Production Manager COTS component provided full services using a variant version of the Production Manager COTS component.

9.6 Summary

This research has been validated through a Distributed Factory Automation System case study. This case study first implemented the COTS component for the Production Manager, the Process Engineer, the Receiving Workstation Controller, the Line Workstation Controller, and the Shipping Workstation Controller. After that, the *COTS component wrappers* for each one of the COTS component detailed above were implemented. In the *COTS component wrapper* implementation, the application-dependent information associated with each one of the COTS component involved has been specified by means of tables. Next, the self-management mechanisms have been evaluated. A series of test cases were developed and executed to determine if the Production Manager COTS component wrapper, the Process Engineer COTS component wrapper, the Receiving Workstation Controller COTS component wrapper, the Line Workstation Controller COTS component wrapper, and the Shipping Workstation Controller COTS component wrapper can detect a malfunction of any of their functions. For these test cases, a fault was inserted in each of the functions provided by each of the COTS component to determine if the *COTS component wrappers* can detect them. Once

a fault has been detected, another series of test cases were performed to verify if the whole system is reconfigured against the malfunction functions. Finally, a last series of test cases were performed to verify if the *COTS component wrappers* can repair the Production Manager COTS components, Process Engineer COTS components, Receiving Workstation Controller COTS component, Line Workstation Controller COTS component, and Shipping Workstation Controller COTS component.

CHAPTER 10

CONTRIBUTIONS AND FUTURE RESEARCH

10.1 Introduction

In this research an approach for self-managing of COTS component-based systems using wrappers has been described. In order to develop the self-managing approach, this research has developed a detection mechanism (chapter 4) to identify anomalies in the COTS components being monitored, a reconfiguration mechanism (chapter 5) to minimize the impact from the anomaly detected, and a repair mechanism (chapter 6) to fix the anomaly in the anomalous COTS component. This research also has designed the COTS component wrapper (chapter 7) to support the three mechanisms described above and has specified a mechanism for the interaction among self-managed COTS component. The self-managing of COTS component-based system using wrappers method has been validated through case studies of a distributed elevator system (chapter 8) and a distributed factory automation system (chapter 9).

This chapter briefly outlines the contributions of this research effort and points to future research possibilities. Section 10.2 describes the contributions of this research. Section 10.3 addresses future research work. Section 10.4 summarizes this chapter.

10.2 Research Contributions

This research has focused on developing an approach for self-managing of COTS component-based systems using wrappers. The main contributions of this research effort are the following:

- a) This research has developed a detection mechanism identifying COTS component anomalies that may result from design errors, unfair resource allocation, or inappropriate controlling of concurrency among threads in the component. The COTS component wrapper monitors the functions provided by a COTS component using statecharts. The statechart makes transition from one state to the other state based on the message received by the COTS component wrapper. If the message does not arrive within a reasonable time interval, the COTS component wrapper detects the anomaly of the function and takes an appropriate action.
- b) This research has also developed a reconfiguration mechanism to minimize the impact of the anomalies detected by isolating the faulty COTS component from other healthy COTS components in the system. As an anomalous COTS component resumes after repair, it also needs to be reconfigured so as to provide full functionality of the system. The reconfiguration of a self-managed COTS component is performed by the COTS component wrapper. A COTS component wrapper contains information about the current configuration of the COTS component, that is, what functions provided by a COTS component are blocked or unblocked due to anomalies within the COTS component and in different COTS components. A COTS component

wrapper also has information about functional dependencies between each function in the component and functions in different components. Using these dependency relationships, the COTS component wrapper generates reconfiguration plans against anomalies within the COTS component and in different COTS components, performing the plans to reconfigure the COTS component by blocking ('Repair Block', 'Malfunction Block', 'Dependency Block', or 'Function Block') or unblocking the interfaces to functions associated with anomalies. Anomalous functions are blocked so that other components cannot request the function paralyzed.

- c) This research has developed a repair mechanism fixing anomalies in the anomalous COTS component. The anomalies in a COTS component are repaired at the level of a component, not at the level of each function of a component. An anomalous COTS component can be re-initialized if the COTS component provides a function for initializing the component state. An anomalous COTS component can also be re-installed using the same version of the COTS component or replaced with a variant of the COTS component if it is available. The variants of a COTS component can be different versions of the COTS component, which may either be implemented differently or provide different performances. Some inputs to a COTS component may need to be modified to fix anomalous functions in the COTS component.
- d) This research has finally designed a wrapper for self-managing COTS component-base systems encapsulating the COTS component to control the interaction between COTS components. A COTS component wrapper of self-

managing COTS components is structured into several objects: COTS Modified Interface, COTS Monitor, Wrapper Controller, Reconfiguration Manager, and Repair Manager. The objects in the *COTS component wrapper* interact with each other to monitor and detect anomalies in a COTS component, dynamically reconfigure the anomalous COTS component and the other COTS components in the system, and repair the anomalous COTS component. By consideration of reusability, application-independent algorithms in the objects of a COTS component wrapper are implemented separately from application-dependent information. Application-independent algorithms of a COTS component wrapper are common to all COTS components so that they are reused for different COTS components. Application-dependent part of a COTS component wrapper contains information specific to a COTS component, which should be updated for a different COTS component.

10.3 Future Research

This section discusses possible future work in the area of self-management of COTS component-bases systems using wrappers.

10.3.1 Enhancements to the COTS Monitor

The capability of decision mechanism in a COTS Monitor needs to be extended to detect anomalies of external devices (such as door or motor in the distributed elevator system) handled by a COTS component. Currently, a COTS Monitor determines if a

function of the COTS component is anomalous or not. However, the anomalies of a function may result from external devices.

10.3.2 Enhancements for Hard Real-Time Systems

The present research of self-managed COTS components has some weaknesses such as degradation of performance. The degradation of performance is due to the time needed to reconfigure and repair a COTS component. During this period of time, the COTS component cannot provide services, or can provide degraded services. Because of this limitation, this approach can be used only to self-manage soft real-time COTS component-systems. Hard real-time systems should provide services within tight deadlines, and the time required for reconfiguring and repairing anomalous functions may go beyond the deadline specified for a particular service. Therefore, this approach needs to be extended to self-managed hard real-time COTS component-systems.

10.3.3 Enhancements to Decrease System Complexity

This research uses many message communications for detecting, reconfiguring, and repairing a COTS component. These message communications may bring additional complexity of the system. The distributed elevator system, implemented as a first case study, has 14 COTS components with their corresponding wrappers for self-management, whereas the distributed factory automation system, implemented as a second case study has 7 COTS components; plus all the connectors needed for communication – inside the wrapper and between COTS components – and all the socket objects needed for communication in a distributed environment. Although we have not found any difficulty

in running the elevator system and the factory automation system, quantitative measurements on complexity of the system are needed so that we provide a way of reducing the complexity.

10.3.4 Extended Validation

Appropriate real COTS components for this research were not available from both open-sources portals and several vendors' web sites because it was hard to find real COTS components for soft real-time systems, and because vendors allow only 30 days to try their COTS components. Furthermore, to properly validate this research we need to have access to the COTS component's implementation in order to induce faults, so that the detection, reconfiguration and repair mechanisms can be evaluated. Therefore, this research has been validated using COTS components developed 'in house'. Even though this approach has been validated using the distributed elevator COTS components-based system, which is developed by a graduate student outside of this research, and the distributed factory automation COTS component-based system developed by me using the design specification presented in Gomaa's book (Gomaa 2000), this approach needs to be validated more using real COTS components systems from the market.

10.4 Summary

This research has presented an approach to designing self-managed COTS component-based systems in which each COTS component is encapsulated in its wrapper that is capable of detecting anomalies in the COTS component, reconfiguring the COTS component against the anomalies detected, and repairing the anomalous COTS

component. To support the capabilities of wrappers for COTS components, the wrappers has been specified and designed. To validate the detection, reconfiguration, and repair mechanisms, and the interactions of the objects constituting the wrapper, two case studies, a distributed elevator system and a distributed factory automation system were implemented.

REFERENCES

- Booch, Grady, James Rumbaugh, and Ivar Jacobson. 2005. *The Unified Modeling Language User Guide*. Reading: Addison-Wesley.
- Buschman, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester, England: John Wiley & Sons Ltd.
- Dashofy, Eric M., André van der Hoek, and Richard N. Taylor. 2002. "Towards Architecture-based Self-Healing Systems". In *First Workshop on Self-Healing Systems: Proceedings of the workshop in Charleston, South Carolina, United States, November 18-19, 2002*.
- DeLine, Robert. 1999. "A Catalog of Techniques for Resolving Packing Mismatch". In *5th Symposium on Software Reusability: Proceedings of the symposium in Los Angeles, California, United States, May 21-23, 1999*, 44-53.
- Garlan, David, Bradley Schmerl. 2002. "Model-based Adaptation for Self-Healing Systems". In *First Workshop on Self-Healing Systems: Proceedings of the workshop in Charleston, South Carolina, United States, November 18-19, 2002*.
- Garlan, David, Shang-Wen Cheng, and Bradley Schmerl. Increasing System Dependability through Architecture-based Self-repair. In *Architecting Dependable Systems*, eds. Rogério de Lemos, Cristina Gacek, and Alexander Rommanovsky, 61-89. New York: Springer Berlin / Heidelberg, 2003.
- Gomaa, Hassan. 2000. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Boston: Addison-Wesley.
- Guerra, Paulo Asterio de C., Cecília Mary F. Rubiera, and Rogério de Lemos. 2002. "An Idealized Fault-Tolerant Architectural Component". In *International Conference on Software Engineering (ICSE'02): Workshop on Architecting Dependable Systems, Orlando, Florida, United States, May 19-25, 2002*.
- Guerra, Paulo Asterio de C., Cecília Mary F. Rubiera, Alexander Romanosvsky, and Rogério de Lemos. 2003. "Integrating COTS Software Components into Dependable Software Architectures". In *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003): Proceedings of the symposium in Hakodate, Hokkaido, Japan, May 14-16, 2003*, 139-142.

- Guerra, Paulo Asterio de C., Cecília Mary F. Rubira, Alexander Romanovsky, and Rogério de Lemos. "A Dependable Architecture for COTS-Based Software Systems using Protective Wrappers". In *Architecting Dependable Systems II. Lecture Notes in Compute Science 3069*, eds. Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky, 144-166, Berlin, Germany: Springer, 2004.
- IBM. 2003. "An architectural blueprint for autonomic computing", *IBM and autonomic computing*. Available from <http://www-03.ibm.com/autonomic/pdfs/ACBlueprintWhitePaperV7.pdf>; Internet; accessed 14 February 2007.
- Kim, K. H. (Kane) and Chittur Subbaraman. Fault-Tolerant Real-Time Objects. *Communications of the ACM* 40, no. 1 (1997): 75-82.
- _____. Dynamic Configuration Management in Reliable Distributed Real-Time Information Systems. *IEEE Transactions on Knowledge and Data Engineering* 11, no. 1 (1999): 239-254.
- Kramer, Jeff and Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering* 16, no. 11 (1990): 1293-1306.
- Koopman, Philip. 2003. "Elements of the Self-Healing System Problem Space". In *Workshop on Software Architectures for Dependable Systems (WADS 2003), International Conference on Software Engineering (ICSE 2003): Proceedings of the conference in Portland, Oregon, United States, May 3-11, 2003*.
- Meulen, Meine van der, Steve Riddle, Lorenzo Strigini, and Nigel Jefferson. 2004. "Protective Wrapping of Off-the-Shelf Components". In *COTS-Based Software System: 4th International Conference, ICCBSS 2005: Proceedings of the conference in Bilbao, Spain, February 7-11, 2005*.
- Rumbaugh, James, Grady Booch, and Ivar Jacobson. 2005. *The Unified Modeling Language Reference Manual*. Reading: Addison-Wesley.
- Shaw, Mary and David Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. New Jersey: Prentice Hall.
- Thomas, Vicraj, SriKumar Kareti, Walter Heimerdinger, and Sunondo Ghosh. 2000. "Mediators and Obligations: An Architecture for Building Dependable Systems Containing COTS Software Components". In *Dependable System Middleware and Group Communication (DSMGC 2000): Proceedings of the International SRDS Workshop in Nuremberg, Germany, October 16-18, 2000*.
- Torres-Pomales, Wilfredo. 2000. "Software Fault Tolerance: A Tutorial". NASA/TM-2000-210616, *ACM Portal*. [e-journal] <<http://portal.acm.org>>.

APPENDIX A

IMPLEMENTATION OF A COTS COMPONENT WRAPPER

A.1 Introduction

This Appendix presents the implementation details of a COTS component wrapper in order to self-manage a COTS component. By consideration of reusability, a *COTS component wrapper* that plays a key role for self-managing COTS components is implemented by means of both application-independent part and application-dependent part. Application-independent part of a *COTS component wrapper* is common to all COTS components so that it is reused for different COTS components once it is implemented. Application-dependent part of a *COTS component wrapper* contains information specific to a COTS component, which should be updated for a different COTS component.

Application-independent part of a COTS component wrapper is algorithms provided by each object of a COTS component wrapper. The COTS Modified Interface, COTS Monitor, Wrapper Controller, Reconfiguration Manager, and Repair Manager object in a COTS component wrapper were implemented using Java programming language. Application-dependent part of a COTS component wrapper is implemented by means of tables. These tables are the result of implementing the entities object described in chapter 7.

This appendix begins by describing the implementation of the detection mechanism in section A.2. Section A.3 addresses the implementation of the reconfiguration mechanism. Finally, section A.4 presents the implementation of the repair mechanism.

A.2 Implementation of the Detection Mechanism

The Detection Mechanism (chapter 4) of a *COTS component wrapper* has been implemented through algorithms provided by the following objects: The COTS Modified Interface and the COTS Monitor. Figure A.1 and A.2 depict the algorithms provided by the COTS Modified Interface and COTS Monitor respectively.

Figure A.1: Algorithms Provided by COTS Modified Interface for Detection Mechanism

Figure A.2: Algorithms Provided by COTS Monitor for Detection Mechanism

The COST Modified Interface object provides the Operation Request application-independent algorithm. The Operation Request algorithm receives the requests from other COTS component in the system, communicating the succession of events to the COTS Monitor object and calling the function requested from the COTS component. The pseudocode for this algorithm is as follows:

```

public OperationRequest( in parameters)
  Extract operationName from the parameters.
  Extract arguments from parameters using Operation Request Parameters Table.
  Show arguments to the console.
  Locate operationName in Operation Status Table
  if (operationName.Status.equals("Unblock"))
    Monitor.ModifiedInterfaceNotification(operationName + "Arrived")
    Create a new thread.
    /* Function call to the corresponding COTS operation */
    return type = operationName(arguments)
    Extract arguments from return type.
    Start the new thread previously created.
    Monitor.ModifiedInterfaceNotification(operationName + "Called")
    Join the new thread with the main thread.
    if (the new thread is not interrupted)
      Locate successorOperation using Successor Operation Table
      if (!successorOperation.equals(null))
        Monitor.ModifiedInterfaceNotification(successorOperation + "Arrived")
      if (!successorObject.equals(null))
        /* Function call to the successor operation */
        successorOperation(parameters)
        Monitor.ModifiedInterfaceNotification(successorOperation + "Called")
    else
      Display error message: "Operation Status is blocked"

```

The COTS Monitor object provides the Modified Interface Notification, Start Timing Operation, and Monitor Time Notification application-independent algorithms. The Modified Interface Notification algorithm receives an event notification from the COTS Modified Interface object, making transition in the statechart for a COTS function in response to the event. The pseudocode for this algorithm is as follows:

```

public ModifiedInterfaceNotification(in operation, event)
  Locate CurrentState in Operation Statechart Information Table using operation
  Locate State in Operation Statechart Table using operation and event
  if (OperationStatechartInformation.CurrentState = OperationStatechart.State)
    OperationStatechartInformation.CurrentState = OperationStatechart.NextState
    if (OperationStatechart.Action = "Start Timing")
      opTime = OperationStatechartInformation.MaximumOperationTime
      StartTimingOperation(opTime)
      Assign current time to OperationStatechartInformation.StartTime
    if (OperationStatechart.Action = "End Timing")
      Assign current time to OperationStatechartInformation.EndTime
    if (OperationStatechartInformation.EndTime –
      OperationStatechartInformation.StartTime >
      OperationStatechartInformation.MaximumOperationTime)
      /* Operation Malfunction, an error is detected
      WrapperController.MalfunctionOperation(operation)
  else
    Display error message: "Event came from a not valid state"

```

The Start Time algorithm measures the time within which the next notification message should arrive from the COTS Modified Interface object. The pseudocode for this algorithm is as follows:

```

private StartTimingOperation(in operation, time)
  Wait the time specified in Time
  MonitorTimeNotification(operation)

```

The Monitor Time Notification algorithm tells the COTS Monitor object that the time specified for the next notification message is over. The pseudocode for this algorithm is as follows:

```

private MonitorTimeNotification(in operation)
  if (OperationStatechart.OperationName.equals(operation))
    if (OperationStatechart.EndTime == 0 )
      /*The Successor Operation has not arrived */
      Controller.MalfunctionOperation(operation)

```

A.3 Implementation of the Reconfiguration Mechanism

The Reconfiguration Mechanism (chapter 5) of a *COTS component wrapper* has been implemented through algorithms provided by the following objects: The Wrapper Controller, Reconfiguration Manager, COTS Modified Interface, and the COTS Monitor. Figure A.3 depicts the algorithms provided by the Wrapper Controller.

<<controller>> Wrapper Controller
<ul style="list-style-type: none"> - MalfunctionOperation(in operation) - ReconfigurationExecutorNotification(in operation, object, action, notification) - MalfunctionOperationOtherComponents(in operation, object, action)

Figure A.3: Algorithms Provided by Wrapper Controller for Reconfiguration Mechanism

The Wrapper Controller object provides the Malfunction Operation, Reconfiguration Executor Notification, and Malfunction Operation Other Component application-independent algorithms. The Malfunction Operation algorithm receives the notification from the COTS Monitor object when a function in a COTS component is malfunction. The pseudocode for this algorithm is as follows:

```

public MalfunctionOperation(in operation)
  Locate the corresponding operation in the Operation Criticality Table
  if (operation.Criticality = "Critical")
    action = "RepairBlock"
  else
    action = "MalfunctionBlock"
  /*Starts the Reconfiguration Process */
  ReconfigurationPlanGenerator.GenerateReconfigurationPlan(operation, selfObject, action)

```

The Reconfiguration Executor Notification algorithm receives a communication from the Reconfiguration Manager object notifying that the reconfiguration process in the COTS component wrapper has been completed successfully. The pseudocode for this algorithm is as follows:

```

public ReconfigurationExecutorNotification (in operation, object, action, notification)
String selfObject = “ ”; /*It is initialized with the name of the COTS component */
if (!operation.equals(null)) /*Operation to be send to other COTS*/
    OtherCOTS.MalfunctionOperationOtherComponent(operation, object, action)
else
    if (notification.equals(“Plan Executed”))
        Locate the operation in the OperationCriticality Table
        if (OperationCriticality.criticality.equals(“Critical”))
            RepairManager.StartRepairProcess(selfObject, “Start Repair Process”);
    else
        if (notification.equals(“Plan After Repair Executed”))
            Show to the Console that the component is in service again.

```

The Malfunction Operation Other Components algorithm receives the malfunction notifications from other *COTS component wrappers* (Wrapper Controller object of other *COTS component wrapper*); therefore, the *COTS component wrapper* receiving the notification needs to be reconfigured based on the malfunction function. The pseudocode for this algorithm is as follows:

```

public MalfunctionOperationOtherComponents(in operation, object, action)
    ReconfigurationPlanGenerator.GenerateReconfigurationPlan(operation, object, action)

```

Figure A.4 shows the algorithms provided by the Reconfiguration Manager object. The Reconfiguration Manager object provides the Generate Reconfiguration Plan, Execute Reconfiguration Plan, Modified Interface Notification, and Monitor Notification application-independent algorithms.

<<manager>> Reconfiguration Manager
- GenerateReconfigurationPlan(in operation, object, action) - ExecuteReconfigurationPlan(in reconfiguration plan) - ModifiedInterfaceNotification(in notification) - MonitorNotification(in notification)

Figure A.4: Algorithms Provided by Reconfiguration Manager for Reconfiguration Mechanism

The Reconfiguration Manager object has been implemented by means of Reconfiguration Plan Generator and Reconfiguration Plan Executor. The Reconfiguration Plan Generator generates a reconfiguration plan against the anomalous function using the Generate Reconfiguration Plan algorithm. The pseudocode for this algorithm is as follows:

```

public GenerateReconfigurationPlan( in operation, in object, in action)
/* Variables used in this algorithm */
boolean signal = false; /* To indicate if operation come from other components */
boolean signal1 = false; /* To add operation status only once to the reconfiguration plan */
String opOperationAux;
String opDependentAux;
String obDependentAux;
String opOperationAux1 = "";
String obDependentAux1 = "";
String opDependenceAux;
String funDependenceAux;
String malFunctionOperation = operation;
String malFunctionObject = object;
String malFunctionAction = action;
String selfObject = "" /*It is initialized with the name of the component */
if (!malFunctionAction.equals("Unblock")) /* Case: Reconfiguration Before Repair */
    Add element to Reconfiguration Plan(malFunctionOperation, "", "")

while (reconfigurationTable has more elements)
    Take the next element from the table
    if (malfunctionAction.equals("RepairBlock")) // Case: Reconfiguration Critical Operation before repair
        if (!signal1)
            Add element to Reconfiguration Plan("allOperations", malFunctionObject, malFunctionAction)
            signal1 = true;
        if (!reconfigurationTable.Operation.equals(opOperationAux1))

```

```

if (!malFunctionAction.equals(OperationStatusTable.status))
    Add element to Reconfiguration Plan(malFunctionOperation, “”, malFunctionAction)
    signal = true;
    opOperationAux1 = reconfigurationTable.Operation;
if (!reconfigurationTable.CallerOperation.equals(null))
    if (!OperationStatusTable.status.equals(“DependencyBlock”))
        opOperationAux = reconfigurationTable.Operation;
        opDependentAux = reconfigurationTable.CallerOperation;
        obDependentAux = reconfigurationTable.CallerComponent;
        if (!opOperationAux.equals(opOperationAux1) &&
            (obDependentAux.equals(obDependentAux1)))
            Add element to Reconfiguration Plan(opOperationAux, obDependentAux,
                “MalfunctionBlock”)
        Add element to Reconfiguration Plan(opDependentAux, “”, “DependencyBlock”)
        obDependentAux1 = obDependentAux;
    else
        Add element to Reconfiguration Plan(malFunctionOperation, malFunctionObject,
            “NotifyConsole”)

if (malFunctionAction.equals(“MalfunctionBlock”)) /*Case: Reconfiguration Non-Critical Operation
    before repair */
    if (!reconfigurationTable.Operation.equals(malFunctionOperation))
    if (!malFunctionAction.equals(OperationStatusTable.status))
        Add element to Reconfiguration Plan(malFunctionOperation, “”, malFunctionAction)
        Add element to Reconfiguration Plan(malFunctionOperation, malFunctionObject,
            malFunctionAction)

    signal = true;
    opOperationAux1 = reconfigurationTable.Operation;
    if (!reconfigurationTable.CallerOperation.equals(null))
    if (!OperationStatusTable.status.equals(“DependencyBlock”))
        opOperationAux = reconfigurationTable.Operation;
        opDependentAux = reconfigurationTable.CallerOperation;
        obDependentAux = reconfigurationTable.CallerComponent;
        if (!opOperationAux.equals(opOperationAux1) &&
            (obDependentAux.equals(obDependentAux1)))
            Add element to Reconfiguration Plan(opOperationAux, obDependentAux,
                “MalfunctionBlock”)
        Add element to Reconfiguration Plan(opDependentAux, “”, “DependencyBlock”)
        obDependentAux1 = obDependentAux;
    else
        Add element to Reconfiguration Plan(malFunctionOperation, malFunctionObject,
            “NotifyConsole”)

if (malFunctionAction.equals(“Unblock”)) /* Case: Reconfiguration after repair */
    if (!signal1)
        Add element to Reconfiguration Plan(“allOperations”, malFunctionObject, malFunctionAction)
        signal1 = true;
    if (!reconfigurationTable.Operation.equals(opOperationAux1))
    if (!malFunctionAction.equals(OperationStatusTable.status))
        Add element to Reconfiguration Plan(reconfigurationTable.Operation, “”,
            malFunctionAction)

    signal = true;
    opOperationAux1 = reconfigurationTable.Operation;
    if (!reconfigurationTable.CallerOperation.equals(null))
    if (!OperationStatusTable.status.equals(“Unblock”))
        opOperationAux = reconfigurationTable.Operation;

```



```

    opDependentAux = reconfigurationTable.CallerOperation;
    obDependentAux = reconfigurationTable.CallerComponent;
    if (!opOperationAux.equals(opOperationAux1) &&
        (obDependentAux.equals(obDependentAux1)))
        Add element to Reconfiguration Plan(opOperationAux, obDependentAux, "Unblock")
    Add element to Reconfiguration Plan(opDependentAux, "", "Unblock")
    obDependentAux1 = obDependentAux;

if (!signal) /* Case: The operation come from other Component */
    if ((reconfigurationTable.Callee.Operation.equals(malFunctionOperation) &&
        (reconfigurationTable.Callee.Component.equals(malFunctionObject))) //Case: Operation Block
        opOperationAux = reconfigurationTable.Operation;
        opDependenceAux = reconfigurationTable.Callee.Operation;
        if (!malFunctionAction.equals("Unblock")) /* Case: Reconfiguration before repair */
            if (!OperationStatusTable.status.equals(malFunctionAction)
                Add element to Reconfiguration Plan(opDependenceAux, "", malFunctionAction)
            if (!OperationStatusTable.status.equals("DependencyBlock"))
                Add element to Reconfiguration Plan(opOperationAux, "", "DependencyBlock")
                Add element to Reconfiguration Plan(opOperationAux, selfObject,
                    "DependencyBlock")
            else /* Case: Reconfiguration after repair */
                if (!OperationStatusTable.status.equals(malFunctionAction))
                    Add element to Reconfiguration Plan(opDependenceAux, "", malFunctionAction)
                if (!OperationStatusTable.status.equals(malFunctionAction))
                    Add element to Reconfiguration Plan(opOperationAux, "", malFunctionAction)
                    Add element to Reconfiguration Plan(opOperationAux, selfObject, malFunctionAction)
        if ((reconfigurationTable.Callee.Function.equals(malFunctionOperation) &&
            (reconfigurationTable.Callee.Component.equals(malFunctionObject))) /*Case: Function Block */
            funDependenceAux = reconfigurationTable.Callee.Function;
            if (!OperationStatusTable.status.equals(malFunctionAction))
                Add element to Reconfiguration Plan(funDependenceAux, "", malFunctionAction)
            if (!malFunctionAction.equals("Unblock")) /* Case: Reconfiguration before repair */
                if (!OperationStatusTable.status.equals("FunctionBlock"))
                    Add element to Reconfiguration Plan(funDependenceAux, selfObject, "FunctionBlock")
            else /* Case: Reconfiguration after repair */
                if (!OperationStatusTable.status.equals(malFunctionAction))
                    Add element to Reconfiguration Plan(funDependenceAux, "", malFunctionAction)
                    Add element to Reconfiguration Plan(funDependenceAux, selfObject, malFunctionAction)
Save Reconfiguration Plan (text format)
ReconfiguratorPlanExecutor.ExecuteReconfigurationPlan(ReconfigurationPlan);

```

The Reconfiguration Plan Executor performs the plan generated by the Reconfiguration Plan Generator using the Execute Reconfiguration Plan algorithm. The pseudocode for this algorithm is as follows:

```

public ExecuteReconfigurationPlan(in ReconfigurationPlan)
String RPE = "" /*It is initialized with the name of the component */
int baPlan = 0 /*1: before Repair plan, 2: after Repair plan */

```

```

while (ReconfigurationPlan has more elements)
    Take the next element from the ReconfigurationPlan
    operation = ReconfigurationPlan.operation;
    object = ReconfigurationPlan.object;
    action = ReconfigurationPlan.action;
    if (action.equals("Unblock"))
        baplan = 2;
    else
        baplan = 1;
    if (object.equals(null) and action.equals(null)) /*Obtain the operation name*/
        malFunctionOperation = operation;
    if (operation.equals("allOperations")) /*RepairBlock or Unblock all operations in COTS*/
        if (!action.equals("Unblock")) /* Case: Before Repair */
            Eliminate thread in Modified Interface Object
            ModifiedInterface.UpdateOperationStatus(operation, action);
        if (action.equals("Unblock")) /* Case: After Repair */
            Monitor.ReinitializeStatechart(operation);
    if (action.equals("Notify Console"))
        Show operation and action to Console
    else
        if (object.equals(RPE) && !operation.equals("allOperations")) /*Operation is provided by the
                                                                    component */
            if (action.equals("MalfunctionBlock") || action.equals("DependencyBlock") ||
                action.equals("Unblock") || action.equals("FunctionBlock"))
                if (action.equals("MalfunctionBlock"))
                    Eliminate thread in Modified Interface Object
                    ModifiedInterface.UpdateOperationStatus(operation, action);
        else
            if (object.equals(null))
                Update OperationStatus.Status with action;
        else
            if (!object.equals(RPE) /* Operation is provided by other component */
                WrapperController.MalfunctionOperationOtherComponents(operation, object, action);

```

The Reconfiguration Plan Executor object also provides the Modified Interface Notification and Monitor Notification algorithms. The Modified Interface Notification algorithm receives the notification from the COTS Modified Interface object once the COTS Modified Interface object has updated the status of the functions indicated in the Reconfiguration Plan. The pseudocode for this algorithm is as follows:

```

public ModifiedInterfaceNotification(in notification)
if (baplan.equals(1)) /* Plan before Repair */
    WrapperController.ReconfigurationExecutorNotification("Plan Executed")
else /* Plan after Repair */
    signal = true; /*To indicate that the object has received communication
                    from the Modified Interface Object */

```

The Monitor Notification algorithm receives the notification from the COTS Monitor object once the COTS Monitor object has reinitialized the statecharts of the functions indicated in the Reconfiguration Plan. This algorithm is executed only after (repair) the COTS Modified Interface has updated the status of the operations as ‘unblock’. This last action is evaluated through a signal, which is initialized with ‘false’ and is changed to ‘true’ when the COTS Modified Interface has performed its work (see `signal = true` in `ModifiedInterfaceNotification` algorithm). The pseudocode for this algorithm is as follows:

```
public MonitorNotification(in notification)
  WrapperController.ReconfigurationExecutorNotification("Plan After Repair Executed")
```

Figure A.5 depicts the algorithm provided by the COTS Modified Interface object.

Figure A.5: Algorithm Provided by COTS Modified Interface for Reconfiguration Mechanism

The COTS Modified Interface object provides the Update Operation Status application-independent algorithm. The Update Operation Status algorithm receives from the Reconfiguration Plan Executor object, the name of the function and the action to be performed over that function, such as ‘repair block’, ‘malfunction block’, ‘dependency block’, ‘function block’, or ‘unblock’. The pseudocode for this algorithm is as follows:

```

public UpdateOperationStatus(in operation, action)
if (operation.equals("allOperations"))
    Update all the OperationStatus.Status elements in the table with action;
else
    Locate the operation in the OperationStatus table
    Update the OperationStatus.status with action
if (action.equals("FunctionBlock"))
    Locate the operation in the FunctionStatus table
    Update the FunctionStatus.status with action
ReconfigurationPlanExecutor.ModifiedInterfaceNotification("Plan Executed")

```

Figure A.6 presents the algorithm provided by the COTS Monitor object. The COTS Monitor object provides the Reinitialize Statechart application-independent algorithm.

Figure A.6: Algorithm Provided by COTS Monitor for Reconfiguration Mechanism

The Reinitialize Statechart algorithm receives from the Reconfiguration Plan Executor object the function whose statechart needs to be reinitialized. The pseudocode for this algorithm is as follows:

```

public ReinitializeStatechart(in operation)
  Locate operation into the OperationStatechartInformation Table
  OperationStatechartInformation.currentState = "Idle";
  OperationStatechartInformation.startTime = 0;
  OperationStatechartInformation.endTime = 0;
  ReconfigurationPlanExecutor.MonitorNotification("Statechart Reinitialized")

```

A.4 Implementation of the Repair Mechanism

The Repair Mechanism (chapter 6) of a *COTS component wrapper* has been implemented through algorithms provided by the following objects: The Repair Manager, COTS Modified Interface, and Wrapper Controller. Figure A.7 shows the algorithms provided by the Repair Manager.

<<manager>> Repair Manager
- StartRepairProcess(in object, notification) - RepairExecuted(in notification)

Figure A.7: Algorithms Provided by Repair Manager for Repair Mechanism

The Repair Manager object provides Start Repair Process and Repair Executed application-independent algorithms. The Start Repair Process algorithm receives from the Wrapper Controller object instructions to start the repair process of the COTS component. The pseudocode for this algorithm is as follows:

```

public StartRepairProcess(in component, notification)
String selfObject = " "; /*It is initialized with the name of the COTS component */
if (component.equals(selfObject) && notification.equals("Start Repair Process"))
  while (RepairTechniques table has more elements)
    Take the next element from the table
    if (RepairTechniques.Status.equals("Non-Used"))
      TechniqueName = RepairTechnique.Technique
      RepairTechnique.Status = "Used";
      break;
  if (TechniqueName = "Re-Install component")
    Create a new object using the same component implementation from a safe location
    ModifiedInterface.RepairManagerNotification(new object, "Re-Install Component")
  if (TechniqueName = "Replace component")
    Create a new object using a variant implementation
    ModifiedInterface.RepairManagerNotification(new object, "Replace Component")

```

The Repair Executed algorithm receives from the COTS Modified Interface object the notification indicating that the COTS Modified Interface object has received the new COTS component and the new COTS component is the one to be used for future requests. The pseudocode for this algorithm is as follows:

```
public RepairExecuted(in notification)
  WrapperController.RepairManagerNotification("Repair Process Executed")
```

Figure A.8 depicts the algorithm provided by the COTS Modified object in support of the repair mechanism. The COTS Modified object provides the Repair Manager Notification application-independent algorithm.

Figure A.8: Algorithm Provided by COTS Modified Interface for Repair Mechanism

The Repair Manager Notification algorithm receives from the Repair Manager object the new COTS component and the action to be performed such as ‘reinstall’ or ‘replace’. The pseudocode for this algorithm is as follows:

```
public RepairManagerNotification(in object, notification)
  Assign object to the corresponding abstract object
  if (notification.equals("Re-Install Component"))
    Assign object to COTS abstract class;
    RepairManager.RepairExecuted("Re-Installation done")
  else
    Assign object to COTS abstract class;
```

RepairManager.RepairExecuted("Replacement done")

Figure A.9 shows the algorithm provided by the Wrapper Controller object. The Wrapper Controller provides the Repair Manager Notification application-independent algorithm.

Figure A.9: Algorithm Provided by Wrapper Controller for Repair Mechanism

The Repair Manager Notification algorithm receives from the Repair Manager object the notification of successful termination of the repair process. The pseudocode for this algorithm is as follows:

```

public RepairManagerNotification(in notification)
String selfObject = ""; /*It is initialized with the name of the COTS component */
if (notification.equals("Repair Process Executed"))
    ReconfigurationPlanGenerator.GenerateReconfigurationPlan("", selfObject, "Unblock")
  
```

APPENDIX B

TESTING OF THE DISTRIBUTED ELEVATOR SYSTEM

B.1 Introduction

This Appendix presents the testing performed on the Scheduler COTS component and Elevator COTS component of the Distributed Elevator System. This appendix begins by describing the testing of the detection mechanism in section B.2. Section B.3 addresses the testing results of the detection mechanism. Section B.4 describes the testing of the reconfiguration mechanism. Section B.5 presents the testing results of the reconfiguration mechanism. Section B.6 shows the testing of the repair mechanism. Finally, section B.7 presents the testing result of the repair mechanism.

B.2 Testing of the Detection Mechanism

To validate the Detection Mechanism in the *Scheduler COTS component*, six text cases were used, two for each function provided by the Scheduler COTS component: Normal case and Fault or error case.

1) serviceRequest function in Scheduler COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Scheduler COTS component preventing the Scheduler COTS component to reply the floor number, direction of the elevator, and the elevator number that

has to satisfy the elevator request to the Scheduler Modified Interface.

In the EleScheduler.java class, the lines 37 to 42, and 45, were eliminated.

2) **updateStatusPlan** function in Scheduler COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Scheduler COTS component preventing the Scheduler COTS component to reply the update acknowledgment to the Scheduler Modified Interface. In the EleStatusPlanServer.java class, the lines 41 to 44 were eliminated.

3) **updateElevatorStatus** function in Scheduler COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Scheduler COTS component preventing the Scheduler COTS component to reply the update acknowledgment to the Scheduler Modified Interface. In the EleStatusPlanServer.java class, the lines 51 to 60 were eliminated.

To validate the Detection Mechanism in the *Elevator COTS component*, fourteen text cases were used, two for each function provided by the Elevator COTS component: Normal case and Fault or error case.

1) **scheduleRequest** function in Elevator COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Elevator COTS component preventing the Elevator COTS component to reply the

floor number and direction of the location of the elevator, the floor number and direction of destination, and the elevator number to the Elevator Modified Interface. In the EleManager.java class, the lines 59 to 77 were eliminated.

2) **sensorInput** function in Elevator COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Elevator COTS component preventing the Elevator COTS component to reply the floor number and direction of the location of the elevator, and the elevator number to the Elevator Modified Interface. In the EleController.java class, the lines 113 to 139, and lines 145 to 150 were eliminated.

3) **runMotor** function in Elevator COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Elevator COTS component preventing the Elevator COTS component to reply the corresponding motor's action to the Elevator Modified Interface. In the Elevator.java class, the line 61 was eliminated.

4) **stopMotor** function in Elevator COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Elevator COTS component preventing the Elevator COTS component to reply the

corresponding motor's action to the Elevator Modified Interface. In the Elevator.java class, the line 74 was eliminated.

5) **openDoor** function in Elevator COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Elevator COTS component preventing the Elevator COTS component to reply the corresponding door's action to the Elevator Modified Interface. In the Elevator.java class, the line 87 was eliminated.

6) **closeDoor** function in Elevator COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Elevator COTS component preventing the Elevator COTS component to reply the corresponding door's action to the Elevator Modified Interface. In the Elevator.java class, the line 100 was eliminated.

7) **elevatorButtonRequest** function in Elevator COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Elevator COTS component preventing the Elevator COTS component to reply the floor number and direction of the location of the elevator, the floor number and direction of requested destinations, and the elevator number to the Elevator Modified Interface. In the EleManager.java class, the lines 41 to 50 were eliminated.

B.3 Testing Result of the Detection Mechanism

The six tests performed on the *Scheduler COTS component* provide the following results:

- 1) **serviceRequest** function in Scheduler COTS Component
 - a. Normal case: the function performed its activity successfully and the Scheduler Modified Interface called the corresponding scheduleRequest function from the corresponding Elevator Modified Interface.
 - b. Fault or error case: the Scheduler Monitor detected the error in the serviceRequest function, and communicated the malfunction to the Scheduler Wrapper Controller who initiated the reconfiguration process.
- 2) **updateStatusPlan** function in Scheduler COTS Component
 - a. Normal case: the function performed its activity successfully updating the overall status plan.
 - b. Fault or error case: the Scheduler Monitor detected the error in the updateStatusPlan function, and communicated the malfunction to the Scheduler Wrapper Controller who initiated the reconfiguration process.
- 3) **updateElevatorStatus** function in Scheduler COTS Component
 - a. Normal case: the function performed its activity successfully updating the elevator status.

- b. Fault or error case: the Scheduler Monitor detected the error in the `updateElevatorStatus` function, and communicated the malfunction to the Scheduler Wrapper Controller who initiated the reconfiguration process.

The fourteen tests performed on the *Elevator COTS component* provide the following results:

1) **scheduleRequest** function in Elevator COTS Component

- a. Normal case: the function performed its activity successfully and the Elevator Modified Interface called the `updateStatusPlan` function from the Scheduler Modified Interface.
- b. Fault or error case: the Elevator Monitor detected the error in the `scheduleRequest` function, and communicated the malfunction to the Elevator Wrapper Controller who initiated the reconfiguration process.

2) **sensorInput** function in Elevator COTS Component

- a. Normal case: the function performed its activity successfully and the Elevator Modified Interface called the `updateElevatorStatus` function from the Scheduler Modified Interface.
- b. Fault or error case: the Elevator Monitor detected the error in the `sensorInput` function, and communicated the malfunction to the Elevator Wrapper Controller who initiated the reconfiguration process.

3) **runMotor** function in Elevator COTS Component

- a. Normal case: the function performed its activity successfully and the Elevator Modified Interface called the motorRun function from the Elevator Motor External Device.
- b. Fault or error case: the Elevator Monitor detected the error in the runMotor function, and communicated the malfunction to the Elevator Wrapper Controller who initiated the reconfiguration process.

4) **stopMotor** function in Elevator COTS Component

- a. Normal case: the function performed its activity successfully and the Elevator Modified Interface called the motorStop function from the Elevator Motor External Device.
- b. Fault or error case: the Elevator Monitor detected the error in the stopMotor function, and communicated the malfunction to the Elevator Wrapper Controller who initiated the reconfiguration process.

5) **openDoor** function in Elevator COTS Component

- a. Normal case: the function performed its activity successfully and the Elevator Modified Interface called the doorOpen function from the Elevator Door External Device.
- b. Fault or error case: the Elevator Monitor detected the error in the openDoor function, and communicated the malfunction to the Elevator Wrapper Controller who initiated the reconfiguration process.

6) **closeDoor** function in Elevator COTS Component

- a. Normal case: the function performed its activity successfully and the Elevator Modified Interface called the doorClose function from the Elevator Door External Device.
- b. Fault or error case: the Elevator Monitor detected the error in the closeDoor function, and communicated the malfunction to the Elevator Wrapper Controller who initiated the reconfiguration process.

7) **elevatorButtonRequest** function in Elevator COTS Component

- a. Normal case: the function performed its activity successfully and the Elevator Modified Interface called the updateStatusPlan function from the Scheduler Modified Interface.
- b. Fault or error case: the Elevator Monitor detected the error in the elevatorButtonRequest function, and communicated the malfunction to the Elevator Wrapper Controller who initiated the reconfiguration process.

B.4 Testing of the Reconfiguration Mechanism

To validate the Reconfiguration Mechanism in the *Scheduler COTS component*, three test cases were performed, one for each function provided by the Scheduler COTS component. Each test case contains the same information as the one used to validate the mechanism in the Floor COTS component.

1) **serviceRequest** function in Scheduler COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Scheduler COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Table B.1 shows the scheduler reconfiguration plan against the anomalous serviceRequest function.

Table B.1: Scheduler Reconfiguration Plan for Service Request Function

Line No.	Operation	Object	Action
1	serviceRequest		
2	allOperations	Scheduler	Repair Block
3	serviceRequest		Repair Block
4	serviceRequest	Floor	Repair Block
5	floorButtonRequest		Dependency Block
6	updateStatusPlan		Repair Block
7	updateStatusPlan	Elevator1	Repair Block
8	scheduleRequest		Dependency Block
9	elevatorButtonRequest		Dependency Block
10	updateStatusPlan	Elevator2	Repair Block
11	scheduleRequest		Dependency Block
12	elevatorButtonRequest		Dependency Block
13	updateStatusPlan	Elevator3	Repair Block
14	scheduleRequest		Dependency Block
15	elevatorButtonRequest		Dependency Block
16	updateElevatorStatus		Repair Block
17	updateElevatorStatus	Elevator1	Repair Block
18	sensorInput		Dependency Block
19	updateElevatorStatus	Elevator2	Repair Block
20	sensorInput		Dependency Block
21	updateElevatorStatus	Elevator3	Repair Block
22	sensorInput		Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1:* This is the name of the function that is malfunction, so the system can identify the reconfiguration plan.

- *Line 2*: This function is critical; therefore, all the functions in Scheduler COTS component must be blocked as ‘Repair Block’. This line is used to communicate only once to the COTS Modified Interface object to repair all the operations provided by the Scheduler COTS component.
 - *Lines 3, 5, 6, 8, 9, 11, 12, 14, 15, 16, 18, 20, and 22*: These lines are used to change the status of the functions in the Scheduler Reconfiguration Operation Status Table in the Reconfiguration Manager object.
 - *Lines 4, 7, 10, 13, 17, 19, and 21*: These functions have dependent operations in Floor COTS component and Elevator COTS components; therefore, those functions have to be blocked as ‘Dependency Block’. These lines are used for the Reconfiguration Plan Executor object to send the malfunction notifications to the Floor Wrapper Controller and Elevator Wrapper Controllers objects.
- c. Reconfiguration in the Floor COTS Component: The Floor Wrapper Controller receives one function that is malfunction. The reconfiguration plan is as follows:
- serviceRequest function: Table B.2 shows the Floor Reconfiguration Plan for the Service Request malfunction function.

Table B.2: Floor Reconfiguration Plan for Service Request Function

Line No.	Operation	Object	Action
1	serviceRequest		
2	serviceRequest		Repair Block
3	floorButtonRequest		Dependency Block
4	floorButtonRequest	Floor	Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1*: This is the function name that is malfunction, so the system can identify the reconfiguration plan.
- *Lines 2 and 3*: These lines are used to change the status of the functions in the Floor Operation Reconfiguration Status Table in the Reconfiguration Manager object.
- *Line 4*: This line is used to communicate to the COTS Modified Interface to ‘Dependency Block’ the function.

d. Reconfiguration in the Elevator COTS Component: The Elevator Wrapper Controller receives two functions that are malfunction. Those reconfiguration plans are as follows:

- `updateStatusPlan` function: Table B.3 shows the Elevator Reconfiguration Plan for the Update Status Plan malfunction function.

Table B.3: Elevator Reconfiguration Plan for Update Status Plan Function

Line No.	Operation	Object	Action
1	updateStatusPlan		
2	updateStatusPlan		Repair Block
3	scheduleRequest		Dependency Block
4	scheduleRequest	Elevator	Dependency Block
5	updateStatusPlan		Repair Block
6	elevatorButtonRequest		Dependency Block
7	elevatorButtonRequest	Elevator	Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1*: This is the function name that is malfunction, so the system can identify the reconfiguration plan.
- *Lines 2, 3, 5 and 6*: These lines are used to change the status of the functions in the Elevator Operation Reconfiguration Status Table in the Reconfiguration Manager object.
- *Lines 4 and 7*: These lines are used to communicate to the COTS Modified Interface to ‘Dependency Block’ the operations.
 - updateElevatorStatus function: Table B.4 shows the Elevator Reconfiguration Plan for the Update Elevator Status malfunction function.

Table B.4: Elevator Reconfiguration Plan for Update Elevator Status Function

Line No.	Operation	Object	Action
1	updateElevatorStatus		
2	updateElevatorStatus		Repair Block
3	sensorInput		Dependency Block
4	sensorInput	Elevator	Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1*: This is the function name that is malfunction, so the system can identify the reconfiguration plan.
 - *Lines 2 and 3*: These lines are used to change the status of the functions in the Elevator Operation Reconfiguration Status Table in the Reconfiguration Manager object.
 - *Lines 4*: This line is used to communicate to the COTS Modified Interface to ‘Dependency Block’ the operation.
- e. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plan before repair showed above. The only difference is that the action to be taken is ‘Unblock’ instead of ‘Repair Block’, ‘Malfunction Block’, or ‘Dependency Block’.

2) **updateStatusPlan** function in Scheduler COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Scheduler COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for serviceRequest function (Table B.1). The only difference is that the line 1 shows updateStatusPlan instead of serviceRequest.

- c. Reconfiguration in the other COTS components: The Floor and Elevators Wrapper Controllers receive the same malfunction function specified in serviceRequest function, generating the same reconfiguration plan depicted in Tables B.2, B.3, and B.4.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is 'Unblock' instead of 'Repair Block', 'Malfunction Block', or 'Dependency Block'.

3) **updateElevatorStatus** function in Scheduler COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Scheduler COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for serviceRequest function (Table B.1). The only difference is that the line 1 shows updateElevatorStatus instead of serviceRequest.
- c. Reconfiguration in the other COTS components: The Floor and Elevators Wrapper Controllers receive the same malfunction function specified in serviceRequest function, generating the same reconfiguration plan depicted in Tables B.2, B.3, and B.4.

- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is ‘Unblock’ instead of ‘Repair Block’, ‘Malfunction Block’, or ‘Dependency Block’.

To validate the Reconfiguration Mechanism in the *Elevator COTS component*, seven test cases were performed, one for each function provided by the Elevator COTS component. Each test case contains the same information as the one used to validate the mechanism in the Floor COTS component.

1) **scheduleRequest** function in Elevator COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Elevator COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Table B.5 shows the elevator reconfiguration plan against the anomalous scheduleRequest function.

Table B.5: Elevator Reconfiguration Plan for Schedule Request Function

Line No.	Operation	Object	Action
1	scheduleRequest		
2	allOperations	Elevator	Repair Block
3	scheduleRequest		Repair Block
4	scheduleRequest	Scheduler	Repair Block
5	serviceRequest		Dependency Block
6	sensorInput		Repair Block
7	sensorInput	Elevator	Notify Console
8	runMotor		Repair Block
9	runMotor	Elevator	Notify Console
10	stopMotor		Repair Block
11	stopMotor	Elevator	Notify Console
12	openDoor		Repair Block
13	openDoor	Elevator	Notify Console
14	closeDoor		Repair Block
15	closeDoor	Elevator	Notify Console
16	elevatorButtonRequest		Repair Block
17	elevatorButtonRequest	Elevator	Notify Console

The explication of each line of the reconfiguration plan follows:

- *Line 1:* This is the name of the function that is malfunction, so the system can identify the reconfiguration plan.
- *Line 2:* This function is critical; therefore, all the functions in Elevator COTS component must be blocked as ‘Repair Block’.
This line is used to communicate only once to the COTS Modified Interface object to repair all the operations provided by the Scheduler COTS component.
- *Lines 3, 5, 6, 8, 10, 12, 14, 15, and 16:* These lines are used to change the status of the functions in the Elevator Reconfiguration Operation Status Table in the Reconfiguration Manager object.

- *Line 4*: This function has dependent operations in Floor COTS component; therefore, that function has to be blocked as ‘Dependency Block’. This line is used for the Reconfiguration Plan Executor object to send the malfunction notifications to the Floor Wrapper Controller object.
 - *Lines 7, 9, 11, 13, 15, and 17*: These functions have no dependent operations in other components and because they are critical, a message should be display to the console.
- c. Reconfiguration in the Floor COTS Component: The Floor Wrapper Controller receives one function that is malfunction. The reconfiguration plan is as follows:
- *scheduleRequest* function: Table B.6 shows the Floor Reconfiguration Plan for the Service Request malfunction function.

Table B.6: Floor Reconfiguration Plan for Schedule Request Function

Line No.	Operation	Object	Action
1	<i>scheduleRequest</i>		
2	<i>scheduleRequest</i>		Repair Block
3	<i>scheduleRequest</i>	Scheduler	Function Block

The explication of each line of the reconfiguration plan follows:

- *Line 1*: This is the function name that is malfunction, so the system can identify the reconfiguration plan.

- *Line 2:* This line is used to change the status of the function in the Floor Operation Reconfiguration Status Table in the Reconfiguration Manager object.
 - *Line 3:* This line is used to communicate to the COTS Modified Interface to ‘Function Block’ the function. The only function that is blocked is the schedule request function provided by the faulty Elevator COTS component. With this ‘function block’ the Scheduler will send a schedule request to one of the other two elevators that are healthy randomly.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plan before repair showed above. The only difference is that the action to be taken is ‘Unblock’ instead of ‘Repair Block’, ‘Malfunction Block’, ‘Dependency Block’ or “Function Block”.

2) **sensorInput** function in Elevator COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Elevator COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for scheduleRequest function (Table B.5). The only difference is that the line 1 shows sensorInput instead of scheduleRequest.

- c. Reconfiguration in the other COTS components: The Floor Wrapper Controller receives the same malfunction function specified in `scheduleRequest` function, generating the same reconfiguration plan depicted in Table B.6.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is ‘Unblock’ instead of ‘Repair Block’, ‘Malfunction Block’, ‘Dependency Block’ or “Function Block”.

3) **runMotor** function in Elevator COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Elevator COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for `scheduleRequest` function (Table B.5). The only difference is that the line 1 shows `runMotor` instead of `scheduleRequest`.
- c. Reconfiguration in the other COTS components: The Floor Wrapper Controller receives the same malfunction function specified in `scheduleRequest` function, generating the same reconfiguration plan depicted in Table B.6.

- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is ‘Unblock’ instead of ‘Repair Block’, ‘Malfunction Block’, ‘Dependency Block’ or “Function Block”.

4) **stopMotor** function in Elevator COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Elevator COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for scheduleRequest function (Table B.5). The only difference is that the line 1 shows stopMotor instead of scheduleRequest.
- c. Reconfiguration in the other COTS components: The Floor Wrapper Controller receives the same malfunction function specified in scheduleRequest function, generating the same reconfiguration plan depicted in Table B.6.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is ‘Unblock’ instead of ‘Repair Block’, ‘Malfunction Block’, ‘Dependency Block’ or “Function Block”.

5) **openDoor** function in Elevator COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Elevator COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for `scheduleRequest` function (Table B.5). The only difference is that the line 1 shows `openDoor` instead of `scheduleRequest`.
- c. Reconfiguration in the other COTS components: The Floor Wrapper Controller receives the same malfunction function specified in `scheduleRequest` function, generating the same reconfiguration plan depicted in Table B.6.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is 'Unblock' instead of 'Repair Block', 'Malfunction Block', 'Dependency Block' or "Function Block".

6) **closeDoor** function in Elevator COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Elevator COTS component needs to be repaired immediately.

- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for `scheduleRequest` function (Table B.5). The only difference is that the line 1 shows `closeDoor` instead of `scheduleRequest`.
- c. Reconfiguration in the other COTS components: The Floor Wrapper Controller receives the same malfunction function specified in `scheduleRequest` function, generating the same reconfiguration plan depicted in Table B.6.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is 'Unblock' instead of 'Repair Block', 'Malfunction Block', 'Dependency Block' or "Function Block".

7) **elevatorButtonRequest** function in Elevator COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Elevator COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for `scheduleRequest` function (Table B.5). The only difference is that the line 1 shows `elevatorButtonRequest` instead of `scheduleRequest`.

- c. Reconfiguration in the other COTS components: The Floor Wrapper Controller receives the same malfunction function specified in `scheduleRequest` function, generating the same reconfiguration plan depicted in Table B.6.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is ‘Unblock’ instead of ‘Repair Block’, ‘Malfunction Block’, ‘Dependency Block’ or “Function Block”.

B.5 Testing Result of the Reconfiguration Mechanism

The three tests performed on the *Scheduler COTS component* provide the following results:

- a) Criticality of the operation: the Scheduler Wrapper Controller had successfully identified the criticality of each operation selecting ‘Critical’ for the three test cases.
- b) Reconfiguration Plan before repair: the Scheduler Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test cases 1 to 3 (Tables B.1, B.2, B.3, and B.4).
- c) Reconfiguration Plan after repair: The Scheduler Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test cases 1 to 3, changing the action to ‘unblock’ (Tables B.1, B.2, B.3, and B.4).

The seven tests performed on the *Elevator COTS component* provide the following results:

- a) Criticality of the operation: the Elevator Wrapper Controller had successfully identified the criticality of each operation selecting ‘Critical’ for the seven test cases.
- b) Reconfiguration Plan before repair: the Elevator Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test cases 1 to 7 (Tables B.5 and B.6).
- c) Reconfiguration Plan after repair: The Elevator Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test cases 1 to 7, changing the action to ‘unblock’ (Tables B.5 and B.6).

B.6 Testing of the Repair Mechanism

To validate the Repair Mechanism in the *Scheduler COTS component*, two test cases were used, one for each technique available to repair the Scheduler COTS component specified in the Scheduler Repair Techniques Table (Table 8.22).

- 1) Re-install component: The technique re-installs the same component using a copy of the Scheduler COTS component kept in a safe place.
- 2) Replace component: The technique selects a variant version of the Scheduler COTS component as a new Scheduler COTS component. To validate this test case, the first technique has been marked as ‘used’, so the algorithm will select this second technique.

To validate the Repair Mechanism in the *Elevator COTS component*, two test cases were used, one for each technique available to repair the Elevator COTS component specified in the Elevator Repair Techniques Table (Table 8.33).

- 1) Re-install component: The technique re-installs the same component using a copy of the Elevator COTS component kept in a safe place.
- 2) Replace component: The technique selects a variant version of the Elevator COTS component as a new Elevator COTS component. To validate this test case, the first technique has been marked as ‘used’, so the algorithm will select this second technique.

B.7 Testing Result of the Repair Mechanism

The two tests performed on the *Scheduler COTS component* provide the following results:

- 1) Re-install component: The Scheduler Repair Manager object selects and executes this technique to repair the Scheduler COTS component successfully. The Scheduler COTS component provides full services using a copy of the same Scheduler COTS component.
- 2) Replace component: The Scheduler Repair Manager object selects and executes this second technique successfully. The Scheduler COTS component provides full services using a variant version of the Scheduler COTS component.

The two tests performed on the *Elevator COTS component* provide the following results:

- 1) Re-install component: The Elevator Repair Manager object selects and executes this technique to repair the Elevator COTS component successfully. The Elevator COTS component provides full services using a copy of the same Elevator COTS component.
- 2) Replace component: The Elevator Repair Manager object selects and executes this second technique successfully. The Elevator COTS component provides full services using a variant version of the Elevator COTS component.

APPENDIX C

TESTING OF THE DISTRIBUTED FACTORY AUTOMATION SYSTEM

C.1 Introduction

This Appendix presents the testing performed on the Process Engineer COTS component, the Receiving Workstation Controller COTS component, the Line Workstation Controller COTS components, and the Shipping Workstation Controller COTS component of the Distributed Factory Automation System. This appendix begins by describing the testing of the detection mechanism in section C.2. Section C.3 addresses the testing results of the detection mechanism. Section C.4 describes the testing of the reconfiguration mechanism. Section C.5 presents the testing results of the reconfiguration mechanism. Section C.6 shows the testing of the repair mechanism. Finally, section C.7 presents the testing result of the repair mechanism.

C.2 Testing of the Detection Mechanism

To validate the Detection Mechanism in the *Process Engineer COTS component*, two test cases were used for the function provided by the Process Engineer COTS component: Normal case and Fault or error case.

- 1) **operationRequest** function in Process Engineer COTS Component
 - a. Normal case: the function performed its activity normally.

- b. Fault or error case: a fault was inserted in the Process Engineer COTS component preventing the Process Engineer COTS component to reply the operation time to the Production Manager COTS component. In the ProcessEngineer.java class, the line 64 was eliminated.

To validate the Detection Mechanism in the *Receiving Workstation Controller COTS component*, six test cases were used, two for each function provided by the Receiving Workstation Controller COTS component: Normal case and Fault or error case.

- 1) **startPart** function in Receiving Workstation Controller COTS Component
 - a. Normal case: the function performed its activity normally.
 - b. Fault or error case: a fault was inserted in the Receiving Workstation Controller COTS component preventing the Receiving Workstation Controller COTS component to reply the successful execution of the function. In the ReceivingWC.java class, the line 65 was eliminated.
- 2) **partPlaced** function in Receiving Workstation Controller COTS Component
 - a. Normal case: the function performed its activity normally.
 - b. Fault or error case: a fault was inserted in the Receiving Workstation Controller COTS component preventing the Receiving Workstation Controller COTS component to reply the process plan ID, the part type ID, the quantity to be manufactured, and the last operation ID to the first Line Workstation Controller COTS component wrapper. In the ReceivingWC.java class, the lines 73, 74, and 82 were eliminated.
- 3) **partRequest** function in Receiving Workstation Controller COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Receiving Workstation Controller COTS component preventing the Receiving Workstation Controller COTS component to reply the successful execution of the function. In the ReceivingWC.java class, the lines 92 and 99 were eliminated.

To validate the Detection Mechanism in the *Line Workstation Controller COTS component*, ten test cases were used, two for each function provided by the Line Workstation Controller COTS component: Normal case and Fault or error case.

1) **partComing** function in Line Workstation Controller COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Line Workstation Controller COTS component preventing the Line Workstation Controller COTS component to reply the successful execution of the function. In the LineWC.java class, the lines 85 to 109, and 117 were eliminated.

2) **partPlaced** function in Line Workstation Controller COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Line Workstation Controller COTS component preventing the Line Workstation Controller COTS component to reply the successful execution of the

function. In the LineWC.java class, the lines 124, 125, and 131 were eliminated.

3) **partAssembled** function in Line Workstation Controller COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Line Workstation Controller COTS component preventing the Line Workstation Controller COTS component to reply the successful execution of the function. In the LineWC.java class, the lines 141 and 143 to 157 were eliminated.

4) **partAssembledPlaced** function in Line Workstation Controller COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Line Workstation Controller COTS component preventing the Line Workstation Controller COTS component to reply the process plan ID, the part type ID, the quantity to be manufactured, and the last operation ID to the Successor Workstation Controller COTS component wrapper. In the LineWC.java class, the lines 170, 175, and 176 were eliminated.

5) **partRequest** function in Line Workstation Controller COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Line Workstation Controller COTS component preventing the Line Workstation Controller COTS component to reply the successful execution of the

function. In the LineWC.java class, the lines 188 and 196 were eliminated.

To validate the Detection Mechanism in the *Shipping Workstation Controller COTS component*, four test cases were used, two for each function provided by the Shipping Workstation Controller COTS component: Normal case and Fault or error case.

1) **partComing** function in Shipping Workstation Controller COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Shipping Workstation Controller COTS component preventing the Shipping Workstation Controller COTS component to reply the successful execution of the function. In the ShippingWC.java class, the line 62 was eliminated.

2) **partPlaced** function in Shipping Workstation Controller COTS Component

- a. Normal case: the function performed its activity normally.
- b. Fault or error case: a fault was inserted in the Shipping Workstation Controller COTS component preventing the Shipping Workstation Controller COTS component to reply the successful execution of the function. In the ShippingWC.java class, the lines 71, 72, and 77 to 96 were eliminated.

C.3 Testing Result of the Detection Mechanism

The two tests performed on the *Process Engineer COTS component* provide the following results:

1) **operationRequest** function in Process Engineer COTS Component

- a. Normal case: the function performed its activity successfully and the Process Engineer Modified Interface sent the corresponding operation execution time to the Production Manager COTS component.
- b. Fault or error case: the COTS Monitor detected the error in the operationRequest function, and communicated the malfunction to the Wrapper Controller who initiated the reconfiguration process.

The six tests performed on the *Receiving Workstation Controller COTS component* provide the following results:

1) **startPart** function in Receiving Workstation Controller COTS Component

- a. Normal case: the function performed its activity successfully and the COTS Modified Interface called the pickPart function from the Pick-and-Place Robot External System.
- b. Fault or error case: the COTS Monitor detected the error in the startPart function, and communicated the malfunction to the Wrapper Controller who initiated the reconfiguration process.

2) **partPlaced** function in Receiving Workstation Controller COTS Component

- a. Normal case: the function performed its activity successfully and the COTS Modified Interface called the partComing function from the first Line Workstation Controller component wrapper.

- b. Fault or error case: the COTS Monitor detected the error in the partPlaced function, and communicated the malfunction to the Wrapper Controller who initiated the reconfiguration process.
- 3) **partRequest** function in Receiving Workstation Controller COTS Component
 - a. Normal case: the function performed its activity successfully and the COTS Modified Interface received acknowledge of successful execution of the function.
 - b. Fault or error case: the COTS Monitor detected the error in the partRequest function, and communicated the malfunction to the Wrapper Controller who initiated the reconfiguration process.

The ten tests performed on the *Line Workstation Controller COTS component* provide the following results:

- 1) **partComing** function in Line Workstation Controller COTS Component
 - a. Normal case: the function performed its activity successfully and the COTS Modified Interface called the pickPart function from the Pick-and-Place Robot External System.
 - b. Fault or error case: the COTS Monitor detected the error in the partComing function, and communicated the malfunction to the Wrapper Controller who initiated the reconfiguration process.
- 2) **partPlaced** function in Line Workstation Controller COTS Component

- a. Normal case: the function performed its activity successfully and the COTS Modified Interface called the assemblePart function from the Assembly Robot External System.
 - b. Fault or error case: the COTS Monitor detected the error in the partPlaced function, and communicated the malfunction to the Wrapper Controller who initiated the reconfiguration process.
- 3) **partAssembled** function in Line Workstation Controller COTS Component
- a. Normal case: the function performed its activity successfully and the COTS Modified Interface received acknowledge of successful execution of the function.
 - b. Fault or error case: the COTS Monitor detected the error in the partAssembled function, and communicated the malfunction to the Wrapper Controller who initiated the reconfiguration process.
- 4) **partAssembledPlaced** function in Line Workstation Controller COTS Component
- a. Normal case: the function performed its activity successfully and the COTS Modified Interface called not only the partComing function from the Successor Workstation Controller COTS component wrapper, but also the partRequest function from the Predecessor Workstation Controller COTS component wrapper.
 - b. Fault or error case: the COTS Monitor detected the error in the partAssembledPlaced function, and communicated the malfunction to the Wrapper Controller who initiated the reconfiguration process.

5) **partRequest** function in Line Workstation Controller COTS Component

- a. Normal case: the function performed its activity successfully and the COTS Modified Interface received acknowledge of successful execution of the function.
- b. Fault or error case: the COTS Monitor detected the error in the partRequest function, and communicated the malfunction to the Wrapper Controller who initiated the reconfiguration process.

The four tests performed on the *Shipping Workstation Controller COTS component* provide the following results:

1) **partComing** function in Shipping Workstation Controller COTS Component

- a. Normal case: the function performed its activity successfully and the COTS Modified Interface called the pickPart function from the Pick-and-Place Robot External System.
- b. Fault or error case: the COTS Monitor detected the error in the partComing function, and communicated the malfunction to the Wrapper Controller who initiated the reconfiguration process.

2) **partPlaced** function in Shipping Workstation Controller COTS Component

- a. Normal case: the function performed its activity successfully and the COTS Modified Interface received acknowledge of successful execution of the function.

- b. Fault or error case: the COTS Monitor detected the error in the partPlaced function, and communicated the malfunction to the Wrapper Controller who initiated the reconfiguration process.

C.4 Testing of the Reconfiguration Mechanism

To validate the Reconfiguration Mechanism in the *Process Engineer COTS component*, one test case was used for the function provided by the Process Engineer COTS component. Each test case contains the same information as the one used to validate the mechanism in the Production Manager COTS component above.

- 1) **operationRequest** function in Process Engineer COTS component
 - a. Criticality of the function: this function is critical for the system; therefore, the Process Engineer COTS component needs to be repaired immediately.
 - b. Reconfiguration Plan before repair: Table C.1 shows the Process Engineer Reconfiguration Plan against the anomalous operationRequest function.

Table C.1: Process Engineer Reconfiguration Plan for Operation Request Function

Line No.	Operation	Object	Action
1	operationRequest		
2	allOperations	Process Engineer	Repair Block
3	operationRequest		Repair Block
4	operationRequest	Production Manager	Repair Block
5	nextOperationRequest		Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1*: This is the name of the function that is malfunction, so the system can identify the reconfiguration plan.
 - *Line 2*: This function is critical; therefore, all the functions in Process Engineer COTS component must be blocked as ‘Repair Block’. This line is used to communicate only once to the COTS Modified Interface object to repair block all the operations provided by the Process Engineer COTS component.
 - *Lines 3 and 5*: These lines are used to change the status of the functions in the Process Engineer Operation Reconfiguration Status Table in the Reconfiguration Manager object.
 - *Line 4*: This function has dependent operations in other COTS components; therefore, this function has to be blocked as ‘Repair Block’. This line is used for the Reconfiguration Plan Executor object to send the malfunction notification to the other Wrapper Controller object.
- c. Reconfiguration in the *Production Manager COTS Component*: The Production Manager Wrapper Controller receives one function that is malfunction. The reconfiguration plan is as follows:
- **operationRequest** function: Table C.2 shows the Production Manager Reconfiguration Plan for the Operation Request malfunction function.

Table C.2: Production Manager Reconfiguration Plan for Operation Request Function

Line No.	Operation	Object	Action
1	operationRequest		
2	operationRequest		Repair Block
3	nextOperationRequest		Dependency Block
4	nextOperationRequest	Production Manager	Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1*: This is the function name that is malfunction, so the system can identify the reconfiguration plan.
 - *Lines 2 and 3*: These lines are used to change the status of the functions in the Production Manager Status Reconfiguration Table in the Reconfiguration Manager object.
 - *Line 4*: This line is used to communicate to the COTS Modified Interface to ‘Dependency Block’ the function.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair showed above. The only difference is that the action to be taken is ‘Unblock’ instead of ‘Repair Block’, ‘Malfunction Block’, or ‘Dependency Block’.

To validate the Reconfiguration Mechanism in the *Receiving Workstation Controller COTS component*, three test cases were used, one for each function provided by the Receiving Workstation Controller COTS component. Each test case contains the

same information as the one used to validate the mechanism in the Production Manager COTS component.

1) **startPart** function in Receiving Workstation Controller COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Receiving Workstation Controller component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Table C.3 shows the Receiving Workstation Controller Reconfiguration Plan against the anomalous startPart function.

Table C.3: Receiving Workstation Controller Reconfiguration Plan for Start Part Function

Line No.	Operation	Object	Action
1	startPart		
2	allOperations	Receiving Workstation Controller	Repair Block
3	startPart		Repair Block
4	startPart	Production Manager	Repair Block
5	createPartOrders		Dependency Block
6	partPlaced		Repair Block
7	createPartOrders		Dependency Block
8	partRequest		Repair Block
9	partRequest	Line 1 Workstation Controller	Repair Block
10	partAssembledPlaced		Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1:* This is the name of the function that is malfunction, so the system can identify the reconfiguration plan.
- *Line 2:* This function is critical; therefore, all the functions in Receiving Workstation Controller COTS component must be

blocked as ‘Repair Block’. This line is used to communicate only once to the COTS Modified Interface object to repair block all the operations provided by the Receiving Workstation Controller COTS component.

- *Lines 3, 5, 6, 7, 8, and 10:* These lines are used to change the status of the functions in the Receiving Workstation Controller Operation Reconfiguration Status Table in the Reconfiguration Manager object.
- *Line 4 and 9:* These functions have dependent operations in other COTS components; therefore, these functions have to be blocked as ‘Repair Block’. These lines are used for the Reconfiguration Plan Executor object to send the malfunction notification to the other Wrapper Controller objects.

c. Reconfiguration in the *Production Manager COTS Component*: The Production Manager Wrapper Controller receives one function that is malfunction. The reconfiguration plan is as follows:

- **startPart** function: Table C.4 shows the Production Manager Reconfiguration Plan for the Start Part malfunction function.

Table C.4: Production Manager Reconfiguration Plan for Start Part Function

Line No.	Operation	Object	Action
1	startPart		
2	startPart		Repair Block
3	createPartOrders		Dependency Block
4	createPartOrders	Production Manager	Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1*: This is the function name that is malfunction, so the system can identify the reconfiguration plan.
- *Lines 2 and 3*: These lines are used to change the status of the function in the Production Manager Operation Reconfiguration Status Table in the Reconfiguration Manager object.
- *Line 4*: This line is used to communicate to the COTS Modified Interface to ‘Dependency Block’ the function.

d. Reconfiguration in the *Line 1 Workstation Controller COTS*

Component: The Line 1 Workstation Controller Wrapper Controller receives one function that is malfunction. The reconfiguration plan is as follows:

- **partRequest** function: Table C.5 shows the Line 1 Workstation Controller Reconfiguration Plan for the Part Request malfunction function.

Table C.5: Line 1 Workstation Controller Reconfiguration Plan for Part Request Function

Line No.	Operation	Object	Action
1	partRequest		
2	partRequest		Repair Block
3	partAssembledPlaced		Dependency Block
4	partAssembledPlaced	Line 1 Workstation Controller	Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1:* This is the function name that is malfunction, so the system can identify the reconfiguration plan.
 - *Lines 2 and 3:* These lines are used to change the status of the functions in the Line 1 Workstation Controller Operation Reconfiguration Status Table in the Reconfiguration Manager object.
 - *Line 4:* This line is used to communicate to the COTS Modified Interface to 'Dependency Block' the function.
- e. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair showed above. The only difference is that the action to be taken is 'Unblock' instead of 'Repair Block', 'Malfunction Block', or 'Dependency Block'.
- 2) **partPlaced** function in Receiving Workstation Controller COTS component
- a. Criticality of the function: this function is critical for the system; therefore, the Receiving Workstation Controller COTS component needs to be repaired immediately.
 - b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for startPart function (Table C.3). The only difference is that the line 1 shows partPlaced instead of startPart.

- c. Reconfiguration in the other COTS components: The Production Manager and Line 1 Workstation Controller Wrapper Controllers receive the same malfunction function specified in startPart function, generating the same reconfiguration plan depicted in Tables C.4 and C.5.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is 'Unblock' instead of 'Repair Block', 'Malfunction Block', or 'Dependency Block'.

3) **partRequest** function in Receiving Workstation Controller COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Receiving Workstation Controller COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for startPart function (Table C.3). The only difference is that the line 1 shows partRequest instead of startPart.
- c. Reconfiguration in the other COTS components: The Production Manager and Line 1 Workstation Controller Wrapper Controllers receive the same malfunction function specified in startPart function,

generating the same reconfiguration plan depicted in Tables C.4 and C.5.

- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is ‘Unblock’ instead of ‘Repair Block’, ‘Malfunction Block’, or ‘Dependency Block’.

To validate the Reconfiguration Mechanism in the *Line Workstation Controller COTS component*, five test cases were used, one for each function provided by the Line Workstation Controller COTS component. Each test case contains the same information as the one used to validate the mechanism in the Production Manager COTS component.

- 1) **partComing** function in Line Workstation Controller COTS component
 - a. Criticality of the function: this function is critical for the system; therefore, the Line Workstation Controller component needs to be repaired immediately.
 - b. Reconfiguration Plan before repair: Table C.6 shows the Line Workstation Controller Reconfiguration Plan against the anomalous **partComing** function.

Table C.6: Line Workstation Controller Reconfiguration Plan for Part Coming Function

Line No.	Operation	Object	Action
1	partComing		
2	allOperations	Line Workstation Controller	Repair Block
3	partComing		Repair Block
4	partComing	Predecessor Workstation Controller	Repair Block
5	partPlaced		Dependency Block
6	partPlaced		Repair Block
7	partPlaced		Dependency Block
8	partAssembled		Repair Block
9	partPlaced		Dependency Block
10	partAssembledPlaced		Repair Block
11	partPlaced		Dependency Block
12	partRequest		Repair Block
13	partRequest	Successor Workstation Controller	Repair Block
14	partAssembled		Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1:* This is the name of the function that is malfunction, so the system can identify the reconfiguration plan.
- *Line 2:* This function is critical; therefore, all the functions in the Line Workstation Controller COTS component must be blocked as ‘Repair Block’. This line is used to communicate only once to the COTS Modified Interface object to repair block all the operations provided by the Line Workstation Controller COTS component.
- *Lines 3, 5, 6, 7, 8, 9, 10, 11, 12, and 14:* These lines are used to change the status of the functions in the Line Workstation Controller Operation Reconfiguration Status Table in the Reconfiguration Manager object.
- *Lines 4 and 13:* These functions have dependent operations in other COTS components; therefore, these functions have to be

blocked as ‘Repair Block’. These lines are used for the Reconfiguration Plan Executor object to send the malfunction notification to the other Wrapper Controller objects.

c. Reconfiguration in the *Predecessor Workstation Controller COTS*

Component: The Predecessor Workstation Controller Wrapper

Controller receives one function that is malfunction. The reconfiguration plan is as follows:

- **partComing** function: Table C.7 shows the Predecessor Workstation Controller Reconfiguration Plan for the Part Coming malfunction function.

Table C.7: Predecessor Workstation Controller Reconfiguration Plan for Part Coming Function

Line No.	Operation	Object	Action
1	partComing		
2	partComing		Repair Block
3	partPlaced		Dependency Block
4	partPlaced	Predecessor Workstation Controller	Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1:* This is the function name that is malfunction, so the system can identify the reconfiguration plan.
- *Lines 2 and 3:* These lines are used to change the status of the functions in the Predecessor Workstation Controller Operation Reconfiguration Status Table in the Reconfiguration Manager object.

- *Line 4:* This line is used to communicate to the COTS Modified Interface to ‘Dependency Block’ the function.
- d. Reconfiguration in the *Successor Workstation Controller COTS Component*: The Successor Workstation Controller Wrapper Controller receives one function that is malfunction. The reconfiguration plan is as follows:
- **partRequest** function: Table C.8 shows the Successor Workstation Controller Reconfiguration Plan for the Part Request malfunction function.

Table C.8: Successor Workstation Controller Reconfiguration Plan for Part Request Function

Line No.	Operation	Object	Action
1	partRequest		
2	partRequest		Repair Block
3	partAssembledPlaced		Dependency Block
4	partAssembledPlaced	Successor Workstation Controller	Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1:* This is the function name that is malfunction, so the system can identify the reconfiguration plan.
- *Lines 2 and 3:* These lines are used to change the status of the functions in the Successor Workstation Controller Operation Reconfiguration Status Table in the Reconfiguration Manager object.

- *Line 4*: This line is used to communicate to the COTS Modified Interface to 'Dependency Block' the function.
- e. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair showed above. The only difference is that the action to be taken is 'Unblock' instead of 'Repair Block', 'Malfunction Block', or 'Dependency Block'.
- 2) **partPlaced** function in Line Workstation Controller COTS component
- a. Criticality of the function: this function is critical for the system; therefore, the Line Workstation Controller COTS component needs to be repaired immediately.
 - b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for partComing function (Table C.6). The only difference is that the line 1 shows partPlaced instead of partComing.
 - c. Reconfiguration in the other COTS components: The Predecessor Workstation Controller and the Successor Workstation Controller Wrapper Controllers receive the same malfunction function specified in partComing function, generating the same reconfiguration plan depicted in Tables C.7 and C.8.
 - d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained

above. The only difference is that the action to be taken is ‘Unblock’ instead of ‘Repair Block’, ‘Malfunction Block’, or ‘Dependency Block’.

3) **partAssembled** function in Line Workstation Controller COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Line Workstation Controller COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for partComing function (Table C.6). The only difference is that the line 1 shows partAssembled instead of partComing.
- c. Reconfiguration in the other COTS components: The Predecessor Workstation Controller and the Successor Workstation Controller Wrapper Controllers receive the same malfunction function specified in partComing function, generating the same reconfiguration plan depicted in Tables C.7 and C.8.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is ‘Unblock’ instead of ‘Repair Block’, ‘Malfunction Block’, or ‘Dependency Block’.

4) **partAssembledPlaced** function in Line Workstation Controller COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Line Workstation Controller COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for partComing function (Table C.6). The only difference is that the line 1 shows partAssembledPlaced instead of partComing.
- c. Reconfiguration in the other COTS components: The Predecessor Workstation Controller and the Successor Workstation Controller Wrapper Controllers receive the same malfunction function specified in partComing function, generating the same reconfiguration plan depicted in Tables C.7 and C.8.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is 'Unblock' instead of 'Repair Block', 'Malfunction Block', or 'Dependency Block'.

5) **partRequest** function in Line Workstation Controller COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Line Workstation Controller COTS component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for partComing function (Table C.6). The only difference is that the line 1 shows partRequest instead of partComing.
- c. Reconfiguration in the other COTS components: The Predecessor Workstation Controller and the Successor Workstation Controller Wrapper Controllers receive the same malfunction function specified in partComing function, generating the same reconfiguration plan depicted in Tables C.7 and C.8.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is ‘Unblock’ instead of ‘Repair Block’, ‘Malfunction Block’, or ‘Dependency Block’.

To validate the Reconfiguration Mechanism in the *Shipping Workstation Controller COTS component*, two test cases were used, one for each function provided by the Shipping Workstation Controller COTS component. Each test case contains the same

information as the one used to validate the mechanism in the Production Manager COTS component.

1) **partComing** function in Shipping Workstation Controller COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Shipping Workstation Controller component needs to be repaired immediately.
- b. Reconfiguration Plan before repair: Table C.9 shows the Shipping Workstation Controller Reconfiguration Plan against the anomalous **partComing** function.

Table C.9: Shipping Workstation Controller Reconfiguration Plan for Part Coming Function

Line No.	Operation	Object	Action
1	partComing		
2	allOperations	Shipping Workstation Controller	Repair Block
3	partComing		Repair Block
4	partComing	Predecessor Workstation Controller	Repair Block
5	partAssembledPlaced		Dependency Block
6	partPlaced		Repair Block
7	partAssembledPlaced		Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1:* This is the name of the function that is malfunction, so the system can identify the reconfiguration plan.
- *Line 2:* This function is critical; therefore, all the functions in the Shipping Workstation Controller COTS component must be blocked as ‘Repair Block’. This line is used to communicate only

once to the COTS Modified Interface object to repair block all the operations provided by the Shipping Workstation Controller COTS component.

- *Lines 3, 5, 6, and 7:* These lines are used to change the status of the functions in the Shipping Workstation Controller Operation Reconfiguration Status Table in the Reconfiguration Manager object.
- *Line 4:* This function has dependent operations in other COTS components; therefore, this function has to be blocked as ‘Repair Block’. This line is used for the Reconfiguration Plan Executor object to send the malfunction notification to the other Wrapper Controller object.

c. Reconfiguration in the *Predecessor Workstation Controller COTS*

Component: The Predecessor Workstation Controller Wrapper Controller receives one function that is malfunction. The reconfiguration plan is as follows:

- **partComing** function: Table C.10 shows the Predecessor Workstation Controller Reconfiguration Plan for the Part Coming malfunction function.

Table C.10: Predecessor Workstation Controller Reconfiguration Plan for Part Coming Function

Line No.	Operation	Object	Action
1	partComing		
2	partComing		Repair Block
3	partAssembledPlaced		Dependency Block
4	partAssembledPlaced	Predecessor Workstation Controller	Dependency Block

The explication of each line of the reconfiguration plan follows:

- *Line 1*: This is the function name that is malfunction, so the system can identify the reconfiguration plan.
- *Line 2 and 3*: These lines are used to change the status of the functions in the Predecessor Workstation Controller Operation Reconfiguration Status Table in the Reconfiguration Manager object.
- *Line 4*: This line is used to communicate to the COTS Modified Interface to ‘Dependency Block’ the function.

d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair showed above. The only difference is that the action to be taken is ‘Unblock’ instead of ‘Repair Block’, ‘Malfunction Block’, or ‘Dependency Block’.

2) **partPlaced** function in Shipping Workstation Controller COTS component

- a. Criticality of the function: this function is critical for the system; therefore, the Shipping Workstation Controller COTS component needs to be repaired immediately.

- b. Reconfiguration Plan before repair: Because this operation is critical and the whole component needs to be repaired, the Reconfiguration Plan is the same as the one stated for partComing function (Table C.9). The only difference is that the line 1 shows partPlaced instead of partComing.
- c. Reconfiguration in the other COTS components: The Predecessor Workstation Controller receives the same malfunction function specified in partComing function, generating the same reconfiguration plan depicted in Table C.10.
- d. Reconfiguration Plan after repair: The reconfiguration plans after repair are the same as the reconfiguration plans before repair explained above. The only difference is that the action to be taken is 'Unblock' instead of 'Repair Block', 'Malfunction Block', or 'Dependency Block'.

C.5 Testing Result of the Reconfiguration Mechanism

The test performed on the *Process Engineer COTS component* provides the following results:

- a) Criticality of the operation: the Process Engineer Wrapper Controller had successfully identified the criticality of each operation selecting 'Critical' for the test case.

- b) Reconfiguration Plan before repair: the Process Engineer Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test case (Tables C.1 and C.2).
- c) Reconfiguration Plan after repair: The Process Engineer Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test case, changing the action to 'unblock' (Tables C.1 and C.2).

The three tests performed on the *Receiving Workstation Controller COTS component* provide the following results:

- a) Criticality of the operation: the Receiving Workstation Controller Wrapper Controller had successfully identified the criticality of each operation selecting 'Critical' for the three test cases.
- b) Reconfiguration Plan before repair: the Receiving Workstation Controller Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test cases 1 to 3 (Tables C.3, C.4, and C.5).
- c) Reconfiguration Plan after repair: The Receiving Workstation Controller Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test cases 1 to 3, changing the action to 'unblock' (Tables C.3, C.4, and C.5).

The five tests performed on the *Line Workstation Controller COTS component* provide the following results:

- a) Criticality of the operation: the Line Workstation Controller Wrapper Controller had successfully identified the criticality of each operation selecting 'Critical' for the five test cases.
- b) Reconfiguration Plan before repair: the Line Workstation Controller Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test cases 1 to 5 (Tables C.6, C.7, and C.8).
- c) Reconfiguration Plan after repair: The Line Workstation Controller Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test cases 1 to 5, changing the action to 'unblock' (Tables C.6, C.7, and C.8).

The two tests performed on the *Shipping Workstation Controller COTS component* provide the following results:

- a) Criticality of the operation: the Shipping Workstation Controller Wrapper Controller had successfully identified the criticality of each operation selecting 'Critical' for the two test cases.
- b) Reconfiguration Plan before repair: the Shipping Workstation Controller Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test cases 1 and 2 (Tables C.9, and C.10).

- c) Reconfiguration Plan after repair: The Shipping Workstation Controller Reconfiguration Manager generated and executed successfully the same reconfiguration plans described in the test cases 1 and 2, changing the action to ‘unblock’ (Tables C.9, and C.10).

C.6 Testing of the Repair Mechanism

To validate the Repair Mechanism in the *Process Engineer COTS component*, two test cases were used, one for each technique available to repair the Process Engineer COTS component specified in the Process Engineer Repair Techniques Table (Table 9.22).

- 1) Re-install component: The technique re-installs the same component using a copy of the Process Engineer COTS component kept in a safe place.
- 2) Replace component: The technique selects a variant version of the Process Engineer COTS component as a new Process Engineer COTS component. To validate this test case, the first technique has been marked as ‘used’, so the algorithm will select this second technique.

To validate the Repair Mechanism in the *Receiving Workstation Controller COTS component*, two test cases were used, one for each technique available to repair the Receiving Workstation Controller COTS component specified in the Receiving Workstation Controller Repair Techniques Table (Table 9.33).

- 1) Re-install component: The technique re-installs the same component using a copy of the Receiving Workstation Controller COTS component kept in a safe place.
- 2) Replace component: The technique selects a variant version of the Receiving Workstation Controller COTS component as a new Receiving Workstation Controller COTS component. To validate this test case, the first technique has been marked as 'used', so the algorithm will select this second technique.

To validate the Repair Mechanism in the *Line Workstation Controller COTS component*, two test cases were used, one for each technique available to repair the Line Workstation Controller COTS component specified in the Line Workstation Controller Repair Techniques Table (Table 9.44).

- 1) Re-install component: The technique re-installs the same component using a copy of the Line Workstation Controller COTS component kept in a safe place.
- 2) Replace component: The technique selects a variant version of the Line Workstation Controller COTS component as a new Line Workstation Controller COTS component. To validate this test case, the first technique has been marked as 'used', so the algorithm will select this second technique.

To validate the Repair Mechanism in the *Shipping Workstation Controller COTS component*, two test cases were used, one for each technique available to repair the

Shipping Workstation Controller COTS component specified in the Shipping Workstation Controller Repair Techniques Table (Table 9.55).

- 1) Re-install component: The technique re-installs the same component using a copy of the Shipping Workstation Controller COTS component kept in a safe place.
- 2) Replace component: The technique selects a variant version of the Shipping Workstation Controller COTS component as a new Shipping Workstation Controller COTS component. To validate this test case, the first technique has been marked as 'used', so the algorithm will select this second technique.

C.7 Testing Result of the Repair Mechanism

The two tests performed on the *Process Engineer COTS component* provide the following results:

- 1) Re-install component: The Process Engineer Repair Manager object selected and executed this technique to repair the Process Engineer COTS component successfully. The Process Engineer COTS component provided full services using a copy of the same Process Engineer COTS component.
- 2) Replace component: The Process Engineer Repair Manager object selected and executed this second technique successfully. The Process Engineer COTS component provided full services using a variant version of the Process Engineer COTS component.

The two tests performed on the *Receiving Workstation Controller COTS component* provide the following results:

- 1) Re-install component: The Receiving Workstation Controller Repair Manager object selected and executed this technique to repair the Receiving Workstation Controller COTS component successfully. The Receiving Workstation Controller COTS component provided full services using a copy of the same Receiving Workstation Controller COTS component.
- 2) Replace component: The Receiving Workstation Controller Repair Manager object selected and executed this second technique successfully. The Receiving Workstation Controller COTS component provided full services using a variant version of the Receiving Workstation Controller COTS component.

The two tests performed on the *Line Workstation Controller COTS component* provide the following results:

- 1) Re-install component: The Line Workstation Controller Repair Manager object selected and executed this technique to repair the Line Workstation Controller COTS component successfully. The Line Workstation Controller COTS component provided full services using a copy of the same Line Workstation Controller COTS component.
- 2) Replace component: The Line Workstation Controller Repair Manager object selected and executed this second technique successfully. The Line

Workstation Controller COTS component provided full services using a variant version of the Line Workstation Controller COTS component.

The two tests performed on the *Shipping Workstation Controller COTS component* provide the following results:

- 1) Re-install component: The Shipping Workstation Controller Repair Manager object selected and executed this technique to repair the Shipping Workstation Controller COTS component successfully. The Shipping Workstation Controller COTS component provided full services using a copy of the same Shipping Workstation Controller COTS component.
- 2) Replace component: The Shipping Workstation Controller Repair Manager object selected and executed this second technique successfully. The Shipping Workstation Controller COTS component provided full services using a variant version of the Shipping Workstation Controller COTS component.