

Software Fault Trees and Software Failure Modes and Effects Approaches for Preliminary Phases of Object-Oriented Software Design

THESIS

**Submitted in partial fulfilment of the requirements for the
degree of**

DOCTOR OF PHILOSOPHY

by

PANKAJ VYAS
(2003PHXF013P)

Under the Supervision of

Prof. R.K. MITTAL



**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI – 333 031 (RAJASTHAN), INDIA**

November 2014



**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI - 333 031 (RAJASTHAN) INDIA**

CERTIFICATE

This is to certify that the thesis entitled “*Software Fault Trees and Software Failure Modes and Effects Approaches for Preliminary Phases of Object-Oriented Software Design*” submitted by PANKAJ VYAS, ID.No. 2003PHXF013P for award of Ph.D. Degree of the institute, embodies original work done by him under my supervision.

Date: _____

Signature: _____

Prof. R K MITTAL

Senior Professor and Director (Special Projects)
BITS, Pilani (Rajasthan)

Dedicated

to my

Parents (Shri Ravi Dutt Sharma & Smt. Satya Rani)

Wife Mrs. Radhna,

Daughter Gayatri (Guddu)

&

Son Ishaan (Kannu)

Acknowledgements

Several people have made significant contributions in the completion of this thesis and who deserve special mention.

First of all I would like to thank my supervisor Prof. R K Mittal (Director, Special Projects, BITS Pilani) for his valuable guidance, encouragement and moral support. It has been a nice experience and a great pleasure to be associated with such a dynamic multi-threaded personality. Besides providing me unflinching encouragement and support in various ways, he has helped me in a great way to improve the quality of the thesis and to enhance and nourish my intellectual growth.

Special thanks are due to Prof. B N Jain (Vice-chancellor, BITS-Pilani, Pilani Campus) and Prof. G. Raghurama (Director, BITS-Pilani, Pilani Campus) for their constant support and encouragement. I am also grateful to Prof. S C Sivasubramanian (Dean, Administration & Chief, CAHU) and Prof. S K Verma (Dean, ARD) for their support.

A special word of appreciation is owed to my DAC members Dr. Yashwardhan Sharma and Dr. N.L.Bhanu Murthy for providing the necessary aid and support on several occasions.

I am greatly indebted to Prof. J P Misra (Chief, IPCU), Prof. S S Balahsubramaniam (Dean, Academics & Resource Planning), Prof. Sudeept Mohan, Prof. Navneet Goyal, Prof. Poonam Goyal, Prof. Mukesh Rohil and Dr. Virender Singh Shekhawat for their guidance, constructive comments, and motivation. I thank them for their willingness to share their knowledge with me, which was very fruitful in shaping my ideas and research. Collective and individual acknowledgements are due to all mycolleagues who have directly or indirectly helped me in my work.

I would also like to thank Mr. Santosh Kumar Saini (Academic Registration & Counseling Division) BITS-Pilani, for his help in compilation of thesis. I also express my thanks to all the staff members of Computer Assisted & Housekeeping Unit (CAHU) and Academic Registration & Counseling Division (ARCD) for their kind support and cooperation towards the completion of my thesis.

Many thanks are due to all my friends for their continuous support and guidance. I thank everybody who was important to the successful realization of this thesis, as well as express my apology that I could not mention personally one by one.

Words fail to express my gratitude to my parents, wife Radhna, daughter Gayatri and son Ishaan, who cheerfully sacrificed the time, which rightfully belonged to them to enable me to complete this study. Without their support, love, care and prayers, this thesis would not have taken this shape.

Finally, I would like to thank God for always guiding me.

PANKAJ VYAS

Software plays a dominant role in safety-critical applications to control and monitor their critical activities. Software safety encapsulates the aspects of software engineering and software assurance that provide a systematic approach to identifying, analyzing, tracking, mitigating, and controlling hazards and hazardous functions of a system where software may contribute either to the hazard or to its mitigation or control, to ensure safe operation of the system (NASA-STD-8719.13C, 2013). The role of software safety is to make sure that software operates within the defined system context and may not cause any unacceptable risk. Software safety analysis is the process of first identifying the potential hazardous states of the system and then providing the mitigation means for the sources of the identified hazards. Two software safety approaches namely Software Fault Tree Analysis (SFTA) (Leveson, 1983a) and Software Failure Modes and Effects Analysis (SFMEA) (Reifer, 1979) are the recommended approaches (NASA-GB-8719.13, 2004) for the analysis of software-induced hazards in the system. SFTA is adapted in software domain by borrowing the features of a hardware safety approach namely Fault Tree Analysis (FTA) (Vesely et al, 1981). Similarly, Software Failure Modes and Effects Analysis (SFMEA) approach is adapted for software by borrowing the features of one another hardware, safety approach, namely Failure Modes and Effects Analysis (FMEA) (MIL-STD-1629A, 1980).

SFTA is a deductive, backward (or top-down) safety analysis approach to the analysis of software induced critical hazards in the system. SFTA approach is backward or top-down in nature because its application starts by first identifying the critical hazardous-state that a system can encounter and then identifying the erroneous events responsible for the occurrence of the identified hazard-state. On the other hand, SFMEA is inductive, forward (or bottom-up) software safety analysis approach and its application first identifies the basic software-related errors that can occur in the system and then investigates the critical effects of these identified errors on the system.

Both SFTA and SFMEA approaches have been explored by researchers in three main software lifecycle phases namely (i) Implementation or Coding, (ii) Requirements Analysis and (iii) Software Design. The inception applications of SFTA and SFMEA approaches are mostly manual, tedious and time-consuming and are directed mainly at

coding phase. However, later on, researchers are successful in making the applications of both these approaches either as semi-automatic or automatic for certain select level of high level languages. For example, Friedmann (Friedmann, 1993) introduced a tool that automatically constructs a software fault-tree for a given Pascal program. Similarly, Ordonio (Ordonio, 1993) introduced an Automated Code Translation Tool (ACTT) to partially automate the software fault tree construction process for Ada programs. Reid (Reid, 1994) and Winter (Winter, 1995) enhanced the features of the ACTT tool by implementing the support for missing Ada structures especially concurrency and exception handling mechanisms. Similarly, the application of SFMEA approach has been automated for Java language (Snooke, 2004; Price and Snooke, 2008; Snooke and Price, 2011).

The current software development techniques are mostly object-oriented based. The Unified Modeling Language (UML) (Booch et al, 2005) is the modeling standard for the software systems developed using object-oriented techniques. The current focus of SFTA and SFMEA research efforts is also directed towards their applications in UML based object-oriented software development process. Various UML models are explored as potential inputs in these application efforts. The objective of these efforts is to make the applications of SFTA and SFMEA approaches as either automatic or semi-automatic. But, these efforts, especially in object-oriented based requirements analysis and design phases, are not successful. The applications of SFTA and SFMEA approaches are still manual and time-consuming in object-oriented based requirements analysis and design phases. This key research issue is addressed in this thesis.

This thesis presents the developed automated and semi-automated SFTA and SFMEA approaches for object-oriented requirements analysis and design phases. The developed SFTA and SFMEA approaches for object-oriented based requirements analysis phase, take the Use-Case Models (UCM) and state diagrams as inputs. The developed SFTA approach for object-oriented requirements analysis phase is automatic whereas the SFMEA approach for object-oriented requirements analysis phase is semi-automatic. The UML sequence and state diagrams are used as inputs in the developed SFTA and SFMEA approaches in object-oriented design phase. The SFTA approach developed for object-oriented design phase is semi-automatic whereas the SFMEA approach developed for object-oriented design phase is automatic. Three software controlled safety-critical case study applications namely (i) Elevator Control System (ECS) (Gomaa, 2005),

(ii) Rail Track Door Control System (RTCS) (Medikonda and Ramaiah, 2010) and (iii) Insulin Delivery System (IDS) (Sommerville, 2005) are used to demonstrate the applications of the developed approaches. The assumptions, relative advantages and limitations of each developed approach are also discussed.

In the final part of this thesis, a novel approach is presented to predict the software reliability of a given use-case functionality during the requirements analysis phase itself. The approach is based upon the applications of the developed SFMEA and SFTA approaches for object-oriented based requirements analysis and design phases.

Table of Contents

S.N.	Description	Page No.
	Acknowledgements	i-ii
	Abstract	iii-v
	Table of contents	vi-x
	List of tables	xi-xiii
	List of figures	xiv-xvi
	List of abbreviations/symbols	xvii
Chapter-1	Introduction	1-16
1.1	GAPS IN RESEARCH	4
1.2	THESIS ORGANIZATION	4
1.3	SOFTWARE FAULT TREE ANALYSIS	6
1.3.1	Qualitative Analysis	6
1.3.2	Quantitative Analysis	8
1.4	SOFTWARE FAILURE MODES AND EFFECTS ANALYSIS	8
1.5	MODELS USED IN UNIFIED MODELING LANGUAGE (UML)	9
1.5.1	Use-Case Models	10
1.5.2	Sequence Diagrams	12
1.5.3	State Charts	15
Chapter-2	Literature Review	17-27
2.1	APPLICATIONS OF SFTA APPROACH	17
2.1.1	In Software Requirements Analysis Phase	17
2.1.2	In Software Design Phase	21
2.2	APPLICATIONS OF SFMEA APPROACH	23
2.2.1	In Software Requirements Analysis Phase	23
2.2.2	In Software Design Phase	25
2.3	RESEARCH GAPS	26

S.N.	Description	Page No.
Chapter-3	Software Fault Tree Analysis Approach in Use-Case based Requirements Analysis Phase	28-74
3.1	PURPOSE OF THE PROPOSED SFTA APPROACH	28
3.2	ASSUMPTIONS FOR THE PROPOSED SFTA APPROACH	29
3.3	HAZARDOUS-STATE DEFINITION	29
3.4	OVERVIEW OF THE PROPOSED SFTA APPROACH	31
3.5	SFTA ALGORITHM	34
3.5.1	Step 1: Extracting Event Sequences for Various Scenarios	34
3.5.2	Step II: Identifying Event-Sequence-State-Transitions for each Scenario	41
3.5.3	Step III: Identifying State-Transition-Errors for all scenarios	43
3.5.4	Step IV: Generating Fault Tree XML File	46
3.5.5	Step V: Drawing Fault Tree From XML File	51
3.5.6	Salient Features and Time Complexity of the SFTA Algorithm	51
3.5.7	Formatting of Inputs	52
3.6	MOTIVATING EXAMPLE 1:REQUEST ELEVATOR USE-CASE OF AN ELEVATOR CONTROL SYSTEM (ECS) APPLICATION	54
3.7	MOTIVATING EXAMPLE 2: RAILWAY TRACK DOOR CONTROL SYSTEM APPLICATION	67
3.8	VALIDATION OF THE ALGORITHM	72
3.9	COMPARATIVE ANALYSIS	74
Chapter-4	Software Failure Modes and Effects Analysis Approach in Use-Case Based Requirements Analysis Phase	75-100
4.1	PURPOSE OF THE PROPOSED SFMEA APPROACH	75
4.2	ASSUMPTIONS FOR THE PROPOSED SFMEA APPROACH	76
4.3	OVERVIEW OF THE PROPOSED SFMEA APPROACH	76
4.4	SFMEA ALGORITHM	78
4.4.1	Step I: Extracting Event-Sequences for each Scenario	78
4.4.2	Step II: Identifying Event-Sequence-State-Transitions For Each Scenario	79

S.N.	Description	Page No.
	4.4.3 Step III: Identifying ‘Event-Errors’ for all Scenarios	79
	4.4.4 Step IV: Performing ‘Event-Errors-Effects-Analysis’ of each Scenario	82
	4.4.5 Time Complexity of the SFMEA Algorithm	84
4.5	APPLICATION OF SFMEA ALGORITHM TO SAFETY-CRITICAL SOFTWARE SYSTEMS	85
	4.5.1 Motivating Example 1: Insulin Delivery System	85
	4.5.2 Motivating Example 2: Railway Track Door Control System (RTCS)	95
	4.5.3 Analysis of Results	97
4.6	COMPARISON OF SFTA AND SFMEA APPROACHES	98
Chapter-5	Software Fault Tree Analysis Approach for Object-Oriented Design Phase	101-131
5.1	OBJECT-ORIENTED DESIGN PROCESS	101
5.2	OBJECTIVE OF THE PROPOSED SFTA APPROACH	102
5.3	ASSUMPTIONS FOR THE PROPOSED APPROACH	102
5.4	OVERVIEW OF THE PROPOSED SFTA ALGORITHM	103
5.5	THE PROPOSED SFTA ALGORITHM	106
	5.5.1 Step I: Extracting Attributes of each Message from a Scenario Sequence Diagram	106
	5.5.2 Step III: Generating Fault Tree XML File for Selected Hazardous-State	106
	5.5.3 Step III: Generating Fault Tree XML File for Selected Hazardous-State	108
	5.5.4 Step IV: Constructing Fault Tree	111
	5.5.5 Time Complexity of the SFTA Algorithm	111
	5.5.6 Formatting of Inputs	112
5.6	APPLICATION OF THE ALGORITHM IN SAFETY-CRITICAL APPLICATION: ELEVATOR CONTROL SYSTEM	114
	5.6.1 Dispatch Elevator Scenario	114
	5.6.2 Stop Elevator Scenario	122
	5.6.3 Analysis of Results	129
5.7	COMPARATIVE ANALYSIS	130

S.N.	Description	Page No.
Chapter-6	Software Failure Modes and Effects Analysis in Object-Oriented Design Phase	132-181
6.1	MOTIVATION FOR SFMEA IN OBJECT-ORIENTED DESIGN PHASE	132
6.2	OVERVIEW OF THE APPROACH	133
6.3	THE PROPOSED SFMEA ALGORITHM	136
6.3.1	Step I: Generating Pseudo Code Form of Sequence Diagram and Extracting Message-Details	137
6.3.2	Step II: Extracting Message-Sequence for each Scenario	143
6.3.3	Step III: Identifying ‘Event-Sequence-State-Transitions’ for each Scenario	147
6.3.4	Perform Message-Errors-Effects-Analysis For Each ‘Event-Sequence-State-Transitions’	152
6.3.5	Time complexity of the Algorithm	161
6.3.6	Sequence and State Diagram Representations	162
6.4	APPLICATION OF SFMEA ALGORITHM TO SAFETY-CRITICAL SOFTWARE SYSTEMS	163
6.4.1	Motivating Example I: Railway Track Door Control System (RTCS)	163
6.4.2	Motivation Example II: Insulin Delivery System (IDS)	171
6.4.3	Analysis of Results	180
6.5	COMPARATIVE ANALYSIS	181
Chapter-7	Software Reliability Prediction for Use-Cases	182-196
7.1	EARLY SOFTWARE RELIABILITY ESTIMATION APPROACHES	182
7.2	PROPOSED SOFTWARE RELIABILITY ESTIMATION APPROACH FOR USE-CASES	184
7.2.1	Assumptions	184
7.2.2	Operational Profile of a Use-Case	184
7.2.3	The Proposed Reliability Estimation Approach for Use Cases	185
7.3	MOTIVATING EXAMPLE 1: INSULIN DELIVERY SYSTEM	187
7.4	MOTIVATING EXAMPLE 2: RAIL TRACK DOOR CONTROL SYSTEM	193
7.5	ANALYSIS OF RESULTS	195
7.6	SOFTWARE RELIABILITY PREDICTION AT OBJECT-ORIENTED DESIGN PHASE	196

S.N.	Description	Page No.
Chapter-8	Conclusion and Future Research Directions	197-201
8.1	PROPOSED SFTA AND SFMEA APPROACHES - SUMMARY	197
8.2	FUTURE RESEARCH DIRECTIONS AND RECOMMENDATIONS	199

References

Appendices (Appendix I to Appendix VIII)

List of publications

Brief biography of the candidate

Brief biography of the supervisor

List of Tables

Table No.	TITLE	Page No.
1.1	Symbols used for Fault Tree Events and Logic Gates	7
1.2	SFMEA Worksheet	9
1.3	A Sample Use Case Realization Template	11
2.1	Summary of SFTA Applications at Requirements Analysis Phase	20
2.2	Summary of SFTA Applications at Design Phase	22
2.3	Summary of SFMEA Applications at Requirements Analysis Phase	24
2.4	Summary of SFMEA Applications at Software Design Phase	26
3.1	Hazardous-State Examples	30
3.2	Structure of Event-Sequence Table	31
3.3	Structure of Event-Sequence-State-Transition	32
3.4	Structure of State-Transition-Error Table	32
3.5	Event-Details of ECS Application	56
3.6	Event-Sequence Table for Scenario 1 of Elevator Control System Application	58
3.7	Event-Sequence Table for Scenario 2 of Elevator Control System Application	58
3.8	Event-Sequence Table for Scenario 3 of Elevator Control System Application	59
3.9	Event-Sequence Table for Scenario 4 of Elevator Control System Application	59
3.10	Event-Sequence Table for Scenario 5 of Elevator Control System Application	59
3.11	Event-Sequence Table for Scenario 6 of Elevator Control System Application	60
3.12	Event-Sequence Table for Scenario 7 of Elevator Control System Application	60
3.13	Event-Sequence-State-Transition Table for Scenario 1	62
3.14	Event-Sequence-State-Transition Table for Scenario	62
3.15	Event-Sequence-State-Transition Table for Scenario 3	63
3.16	Event-Sequence-State-Transition Table for Scenario 4	63
3.17	Event-Sequence-State-Transition Table for Scenario 5	63
3.18	Event-Sequence-State-Transition Table for Scenario 6	64
3.19	Event-Sequence-State-Transition Table for Scenario 7	64
3.20	State-Transition-Error Table for Elevator Control Application	64
3.21	Event-Details of RTCS Application	68
3.22	Event-Sequence table for Scenario 1	69
3.23	Event-Sequence table for Scenario 2	69
3.24	Event-Sequence-State-Transition table for Scenario 1	70
3.25	Event-Sequence-State-Transition table for Scenario 2	70
3.26	State-Transition-Error Table	71

Table No.	TITLE	Page No.
4.1	Structure of Event-Errors	77
4.2	Structure of 'Event-Errors-Effects-Analysis'	77
4.3	Event-Errors For Scenario 1 and Scenario 2	81
4.4	Event-Errors-Effects-Analysis Scenario 1	84
4.5	Event-Details Table For Insulin Delivery System	86
4.6	Event-Sequence Table for Scenario 1 of Insulin Delivery System	87
4.7	Event-Sequence Table for Scenario 2 of Insulin Delivery System	87
4.8	Event-Sequence-State-Transition Table for Scenario 1	89
4.9	Event-Sequence-State-Transition Table for Scenario 2	89
4.10	Event-Errors for Insulin Delivery System	90
4.11	Event-Errors-Effects-Analysis Table for Scenario 1	91
4.12	Event-Errors-Effects-Analysis Table for Scenario 2	94
4.13	Event-Errors Identified For RTCS Application	95
4.14	Event-Errors-Effects-Analysis For Scenario 1	96
4.15	Event-Errors-Effects-Analysis For Scenario 2	96
5.1	Attributes Extracted From Messages	104
5.2	Structure of 'Event-Sequence-State-Transition' Table	104
5.3	Message-Sequence Table Generated for Dispatch Elevator Scenario	116
5.4	Event-Sequence-State-Transition Table Generated for Dispatch Elevator Scenario	119
5.5	Message-Sequence Table Generated for Stop Elevator Scenario	124
5.6	Event-Sequence-State-Transition Table Generated for Stop Elevator Scenario	126
6.1	Various Attributes of a Message	133
6.2	Structure of Message-Sequence	134
6.3	Structure of Event-Sequence-State-Transition	134
6.4	Structure of Message-Errors-Effects-Analysis	135
6.5	Pseudo Code Forms of Various Interaction Operators	137
6.6	Message-Details Extracted from Sequence Diagram of Figure 6.2	142

Table No.	TITLE	Page No.
6.7	Message-Sequence For Scenario 1	147
6.8	Message-Sequence For Scenario 2	147
6.9	Classification of Message-Related Error Categories	152
6.10	Message-Errors-Effects-Analysis For Scenario 1	157
6.11	Message-Errors-Effects-Analysis For Scenario 2	158
6.12	Message-Details Created For Rail Track Door Controller Application	165
6.13	Message-Sequence For Scenario 1 of RTCS Application	166
6.14	Message-Sequence For Scenario 2 of RTCS Application	166
6.15	'Event-Sequence-State-Transitions' for Scenario 1 of RTCS Application	168
6.16	'Event-Sequence-State-Transitions' for Scenario 2 of RTCS Application	168
6.17	Message-Errors-Effects-Analysis For Scenario 1 of RTCS Application	169
6.18	Message-Errors-Effects-Analysis For Scenario 2 of RTCS Application	170
6.19	Message-Details Extracted For Insulin Delivery System	173
6.20	Message-Sequence For Scenario 1 of IDS Application	174
6.21	Message-Sequence For Scenario 2 of IDS Application	175
6.22	'Event-Sequence-State-Transitions' for Scenario 1 of IDS Application	177
6.23	'Event-Sequence-State-Transitions' for Scenario 2 of IDS Application	177
6.24	Message-Errors-Effects-Analysis For Scenario 1 of IDS Application	178
6.25	Message-Errors-Effects-Analysis For Scenario 2 of IDS Application	179
7.1	Assumed Probability of Occurrence of Event-Related Errors of IDS Application	189
7.2	Probability of Occurrence of Event-Related Errors of RTCS Application	194

List of Figures

Figure No.	TITLE	Page No.
1.1	A Sample Fault Tree for Hazardous-state X	7
1.2	A Sample Use Case Model	10
1.3	A Sample Sequence Diagram	12
1.4	Sequence Diagram Example representing an ‘alt’ block	13
1.5	Sequence Diagram with an ‘opt’ block	14
1.6	Sequence Diagram with a ‘break’ block	14
1.7	Sequence Diagram with a ‘loop’ block	15
1.8	Sequence Diagram with a ‘loop’ block	16
1.9	A Simple UML State Diagram	16
1.10	UML State Diagram with Concurrent Sub-states	16
3.1	Overview of the Proposed SFTA Algorithm	33
3.2	An Example Use-Case Model	34
3.3	Use Case Description File For the ‘doOperation’ Use-Case of Figure 3.2	34
3.4	Operational Details of Step I(a)	36
3.5	Scenario Extraction (Step I(b)) for Use-Case Description of Figure 3.3	39
3.6	Scenario Extraction Process Illustration for Use-Case Description Shown in (a)	40
3.7	State Diagrams for two Components X and Y	42
3.8	Event-Sequence-State-Transition Tables for Two Scenarios	42
3.9	Populating State-Transition-Error Table Generated From Tables of Figure 3.8	45
3.10	Illustration of Software Fault Tree Construction Process	48
3.11	Use Case Description File for ‘Request Elevator’ Use-Case of an ECS Application	55
3.12	Door State Diagrams for Elevator Control System Application	61
3.13	Motor State Diagrams for Elevator Control System Application	61
3.14	faulttree.xml file for Hazardous-State: Door!=closed AND Motor=moving	65

Figure No.	TITLE	Page No.
3.15	Fault Tree for Hazardous State Door! = closed AND Motor = moving	65
3.16	faulttree.xml file for Hazardous-State: Motor !=stopped AND Door = opened	66
3.18	faulttree.xml file for Hazardous-State: Door=opened AND Motor = moving	67
3.19	Fault Tree for Hazardous-State Door = opened AND Motor = moving	67
3.20	Use Case Description File for ‘Open Rail Track Door’ Use-Case of RTCS Application	68
3.21	Input State Diagrams for Rail Track Door Control System Application	70
3.22	faulttree.xml file for Hazardous-State: Track_Door !=closed AND Track_Signal = green	71
3.23	Fault Tree Generated For Hazardous-State Track_Door = closed AND Track_Signal = green	72
4.1	Overview of the Proposed SFMEA approach	78
4.2	Use Case Description File for ‘Deliver Insulin’ Use-Case of IDS	86
4.3	State Diagrams for Insulin Delivery System	88
4.4	Mapping of Event-Related Errors and Erroneous State Level Effects of RTCS	98
5.1	Overview of the Proposed SFTA Algorithm for Object-Oriented Design Phase	105
5.2	A Sample Sequence Diagram using ‘alt’ block	113
5.3	Elevator Controller Sequence Diagram (Dispatch Elevator Scenario)	115
5.4	State Diagrams For Dispatch Elevator Scenario of ECS Application	117
5.5	faulttree.xml File for Case 1 Hazardous-State Door((Device)) != closed and Motor((Device)) = moving	120
5.6	Fault Tree Generated for Case 1 Hazardous-State Door((Device))!= closed and Motor((Device)) = moving	120
5.7	faulttree.xml File for Case 3 Hazardous-State Door((Device)) != closed and Motor((Device)) != moving	121
5.8	Fault Tree generated for Case 3 Hazardous-State Door((Device)) != closed and Motor((Device)) != moving	121
5.9	Sequence Diagram for Stop Elevator Scenario	123

Figure No.	TITLE	Page No.
5.10	State Diagrams for Stop Elevator Scenario	125
5.11	Fault Tree XML File for Case 1 Hazardous-State 'Motor((Device))!= stopped and Door((Device))=opened'	127
5.12	Fault Tree Generated for Case 1 Hazardous-State 'Motor((Device))!= stopped and Door((Device))=closed'	128
5.13	faultree.xml File for Case 2 Hazardous-State Motor((Device)) = stopped and Door((Device)) != opened	128
5.14	Fault Tree Generated for Case 1 Hazardous-State Motor((Device)) = stopped and Door((Device)) != closed	129
6.1	Overview of the Proposed SFMEA Approach	135
6.2	A Simple Sequence Diagram With Nested Numbering Scheme	136
6.3	Generating Pseudo Code Form From Sequence Diagram	139
6.4	Message-Sequence-Control-Flow-Graph Constructed For Pseudo Code Description of Figure 6.3(b)	144
6.5	State Diagrams for the Participating Objects	149
6.6	Execution of Step III of The Proposed Approach For Scenario 1	150
6.7	Execution of Step III of The Proposed Approach For Scenario 2	151
6.8	Message Sequence Diagram For Rail Track Door Controller System	163
6.9	Pseudo Code Form of the Sequence Diagram of Figure 6.13	164
6.10	State Diagrams for Rail Track Door Controller Application	167
6.11	Sequence Diagram for Insulin Delivery Pump System	172
6.12	Pseudo Code Form of the Sequence Diagram of Figure 6.13	174
6.13	State Diagrams For The Participating Object of IDS System	176
7.1	An Example Use-Case Model	184
7.2	Fault Tree for Failure of Scenario 1 of IDS Application	191
7.3	Fault Tree for Failure of Scenario 2 of IDS Application	192
7.4	Fault Tree for Failure in Scenario 1 of RTCS Application	195
7.5	Fault Tree for Failure in Scenario 2 of RTCS Application	195

List of Abbreviations/Symbols

TERM	DEFINITION
ACTT	Automated Code Translation Tool
ATM	Automated Teller Machine
BBS	Brake-by Wire System
BDSA	Bi-Directional Safety Analysis
CBUM	Component-Based UML Model
CSDM	Common Safety Description Model
CVA	Commonality and Variability Analysis
DFTs	Dynamic Fault Trees
ECS	Elevator Control System
FFA	Function Failure Analysis
FMEA	Failure Modes and Effects Analysis
FTA	Fault Tree Analysis
HazOp	Hazard and Operability
IDS	Insulin Delivery System
RTCS	Rail Track Door Control System
SFMEA	Software Failure Modes and Effects Analysis
SFTA	Software Fault Tree Analysis
SRFT	Software Requirement Fault Tree
SST	Software Success Tree
UAV	Unpiloted Aerial Vehicle
UCM	Use Case Model
UCRT	Use Case Realization Template
UML	Unified Modeling Language

CHAPTER 1

Introduction

Software is being used extensively these days in controlling and monitoring the functionalities of many safety-critical systems. Software has been employed in this role to control and monitor the working of these systems in an unambiguous, hazard-free and more reliable manner. However, the experience observed in the last three decades is not up to the expectations of the software developers. For example, software induced control errors have been found to be the main reasons behind the occurrence of two famous catastrophic events such as Threac25 (Leveson, 1993) and Ariane5 (Lann, 1997) and more recent incidents are CryosatRocket failure (Cryosat_Rocket_Fault, 2005) and QantasFlight failure (Qantas_Flight_72, 2008). Moreover, simple searches query ‘Software Related Failures’ on a search engine returns the links of many more similar examples. Hence, improving the quality of safety-critical software - particularly enhancing its software safety (Leveson and Harvey, 1984, 1986) aspect, has become a prime concern for software developers of safety-critical system.

The role of software safety is to make sure that software will operate within the defined system context and will not cause any unacceptable risk. Software safety aspect of the system is generally improved and enhanced by employing two types of software failure analysis methods. The first method starts the software failure analysis process by first identifying the various software related errors that can occur in the system during operation and then investigating the catastrophic effects (hazardous-states) that may be caused because of those errors (if the errors go undetected and without any safeguards) on the system. On the other hand, the second method starts the analysis by first identifying the critical hazardous-states that a system can encounter and then finding out the software related errors responsible for the occurrence of these hazardous-states. The first form of safety analysis is known as a *forward-safety analysis* (bottom-up i.e. errors to hazard analysis) whereas the second form is known as *backward-safety analysis* (top-down i.e. hazard to error analysis).

Over the years, researchers are applying traditional well-tested, well-documented and standardized hardware safety analysis and reliability estimation techniques to software.

Software Fault Tree Analysis (SFTA – a Top Down or backward safety analysis) (Taylor, 1982; Leveson and Harvey, 1983a,1983b) and Software Failure Modes and Effects Analysis (SFMEA- Bottom up or forward safety analysis) (Reifer,1979; Georgieva, 2010; Stadler and Seidl,2013) are such techniques adapted from the hardware domain for the analysis of software related erroneous events. SFTA is an extension of a hardware safety analysis technique named Fault Tree Analysis (FTA) (Vesely et al, 1981) to software. Similarly, SFMEA is an extension of hardware safety analysis approach named Failure Modes Effects Analysis (FMEA) (MIL-STD-1629A,1980) to software. The applications of these approaches are complementary in nature because of their backward and forward analysis methods and when used together they generally augment the results obtained by the application of the other approach. For example, the application of SFMEA can help to identify the effects of the software related errors, which have been missed during the application of SFTA approach. Similarly, the application of SFTA can help to identify the basic software errors, which may have been missed during the application of SFMEA approach. National Aeronautics and Space Administration (NASA) recommend to apply both these approaches (NASA-GB-8719.13, 2004), especially in three phases, namely requirements analysis, software design and coding/implementation phases of software life cycle in order to improve the overall robustness of the system.

Over the years, the researchers have either explored the effective strengths of both SFTA and SFMEA approaches, via standalone or integrated applications, in almost every phase of software life cycle. The current focus of both, SFTA and SFMEA research efforts, have been directed towards their integrated application in the early phases of software development lifecycle namely requirements analysis and design phases of object-oriented software development process. Object-oriented techniques and their associated industry modeling standard Unified Modeling Language (UML) (Booch et al, 2005), have revolutionized the software development process and their use has even started for developing safety-critical applications as well. That is why, both SFTA and SFMEA approaches are being explored by the research community for their effectiveness in object-oriented development process.UML provides the modeling support in every phase of object-oriented software development process. The UML models are used for the applications of SFTA and SFMEA approaches in various phases of object-oriented software development. For example, use-case models (UCM) (Jacobson et al, 1992;

Cockburn, 2000) are used as a standard requirement modeling tool and have been used for applying SFTA and SFMEA approaches in object-oriented requirements analysis phase (Balz and Goll, 2005; Douglass, 2009; Troubitsyna, 2011; Gupta et al, 2012). Similarly, sequence and state diagrams are used to depict the dynamic/functional/behavioral aspects of the system and have been used for the applications of both SFTA and SFMEA approaches in the object-oriented software design phase (Pai and Dugan, 2002; Towhidnejad, 2003; David et al, 2008; Kim et al, 2010).

The inception applications of both SFTA and SFMEA approaches were manual, tedious and time-consuming and are directed mainly at coding phase (Reifer, 1979; Taylor, 1982; Leveson and Stolzy, 1983; Cha et al, 1988; Leveson et al, 1991). Later on, researchers have been successful in making the applications of both these approaches either semi-automatic or automatic for some high level languages. For example, Friedmann (Friedmann, 1993) described a tool that automatically constructs a software fault-tree for a given Pascal program. Ordonio (Ordonio, 1993) described an Automated Code Translation Tool (ACTT) to partially automate the software fault tree construction process for Ada programs. Reid (Reid, 1994) and Winter (Winter, 1995) enhanced the features of the ACTT tool by implementing the support for missing Ada structures especially concurrency and exception handling mechanisms. Similarly, the application of SFMEA approach has been automated for Java language (Snooke, 2004; Price and Snooke, 2008; Snooke and Price, 2011).

Currently, the available methodologies for applications of SFTA and SFMEA approaches in object-oriented requirements analysis and design phases are not only manual but also without any systematic method. Many researchers are making efforts either to semi-automate or fully automate the applications of these approaches. However, these efforts have not been reported as fully successful so far. Especially, in use-case based requirements analysis phase and in UML based modeling phase, the applications of both these approaches are still manual, time-consuming and hence error prone. A complete update of review of the literature is given in Chapter 2.

This thesis is the documentation of the research efforts in applying SFTA and SFMEA approaches in the early phases of the software development for safety critical software systems and automating the processes to the extent possible for safety analysis and reliability estimation. The thesis presents the novel approaches developed to automate or semi-automate the applications of SFTA and SFMEA approaches in both object-oriented

requirements analysis and object-oriented design phases. The novel algorithms are developed using UML use-case models for the applications of both approaches in object-oriented requirements analysis phase. Similarly, algorithms are developed using UML sequence and state diagrams for the applications of these approaches in object-oriented design phase.

Software reliability is defined as the probability of failure-free operation of a software system for a specified time in a specified environment. Generally, the reliability of a software system is assessed during the testing phase or more specifically after the implementation phase. The review of the literature for software reliability indicates that there exist few approaches, which can be used to estimate the software reliability during the early software development phases, especially in requirements analysis and design phases. This thesis aims to use the applications of SFTA and SFMEA approaches for estimating the reliability of object-oriented software systems during requirements analysis and design phases and presents the use of results of both SFTA and SFMEA application approaches for early software reliability estimation.

1.1 GAPSIN RESEARCH

As mentioned earlier, that there are ongoing efforts among the researchers to automate or semi-automate the applications of SFTA and SFMEA approaches in object-oriented based requirements analysis and design phases. In order to address this key research issue, this thesis aims to achieve the following research objectives.

1. Developing algorithms for automatic or semi-automatic application of SFTA and SFMEA approaches in Use-Case based requirements analysis phase.
2. Developing algorithms for automatic or semi-automatic applications of SFTA and SFMEA approaches in UML based object-oriented design phase.
3. Developing a SFTA and SFMEA based approach for early reliability prediction of use-case functionality.

1.2 THESIS ORGANIZATION

In order to achieve the above-mentioned research objective, the thesis is organized as follows.

The rest of this first chapter gives a brief introduction of the fundamentals of SFTA and SFMEA approaches respectively. This is followed by the introduction of three UML

models named use case models, sequence diagrams and state diagrams in object-oriented software development process.

Chapter 2 presents the up to date review of the published literature on the use of SFTA and SFMEA approaches in the software requirement analysis and design phases. The research gaps in the applications of the SFTA and SFMEA approaches, especially in the requirements analysis and design phases, are presented at the end of the Chapter.

Chapter 3 presents a new automated SFTA approach for use-case based requirement analysis phase. The formal description of a given use-case functionality and the state diagrams of the participating components are used as inputs. The approach is validated via its applications on the use-case functionalities selected from two software controlled safety-critical applications namely (i) Elevator Control System (ECS) and (ii) Rail Track Door Control System (RTCS).

Chapter 4 presents a new semi-automated SFMEA approach for object-oriented requirements analysis phase. The approach is developed to overcome some of the limitations of the SFTA approach of Chapter 3. The strengths of the developed SFMEA approach are demonstrated by applying it on two safety-critical case study applications namely (i) Insulin Delivery System (IDS) and (ii) Rail Track Door Control System (RTCS).

Chapter 5 describes the proposed semi-automated SFTA approach for UML based object-oriented design phase. The UML sequence diagram drawn for a given use-case functionality and the state diagrams of the collaborating objects are used as inputs. The hazardous-states, for which the fault trees are constructed, are selected from an ECS application.

Chapter 6 describes the proposed automated SFMEA approach for object-oriented design phase. The approach is developed to overcome some of the limitations of the SFTA approach of Chapter 5.

The relative merits and limitations of the proposed SFTA and SFMEA approaches are discussed at the end of Chapters 3 to Chapter 6.

Chapter 7 describes a SFMEA and SFTA based early software reliability prediction approach for use-case based requirements analysis phase. The approach is applied on the use-cases selected from IDS and RTCS applications. The comparative analysis of the proposed approach with other similar approaches is presented at the end of the chapter.

A summary of the work done for this thesis, namely the SFTA and SFMEA approaches developed for object-oriented based requirements analysis and design phases, is given in Chapter 8. The directions for carrying out further research, in order to overcome the shortcomings of the developed approaches, are also outlined.

1.3 SOFTWARE FAULT TREE ANALYSIS


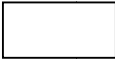


SFTA is a deductive safety analysis approach for the analysis of software induced critical hazards in the system. The approach is backward or top-down in the sense that its application first selects a critical hazardous-state that a system can encounter and then recursively traces its causes in backward direction either in code or in design or in specified requirements, to identify all the logical combinations of software related errors that contribute towards the occurrence of the selected critical hazardous-state until the basic software related errors are reached. The root node event (hazard-state) and the basic error events are joined by suitable events and gate symbols. The application of SFTA approach results in a tree like graphical structure known as - *software fault tree* whose root node represents the specified critical hazardous-state and the leaf nodes represent the identified basic software related errors. The symbols used for the events and gates, that are used to draw the software fault tree, are shown in Table 1.1 and a simple fault tree with four basic events and three logical gates is shown in Figure 1.1.

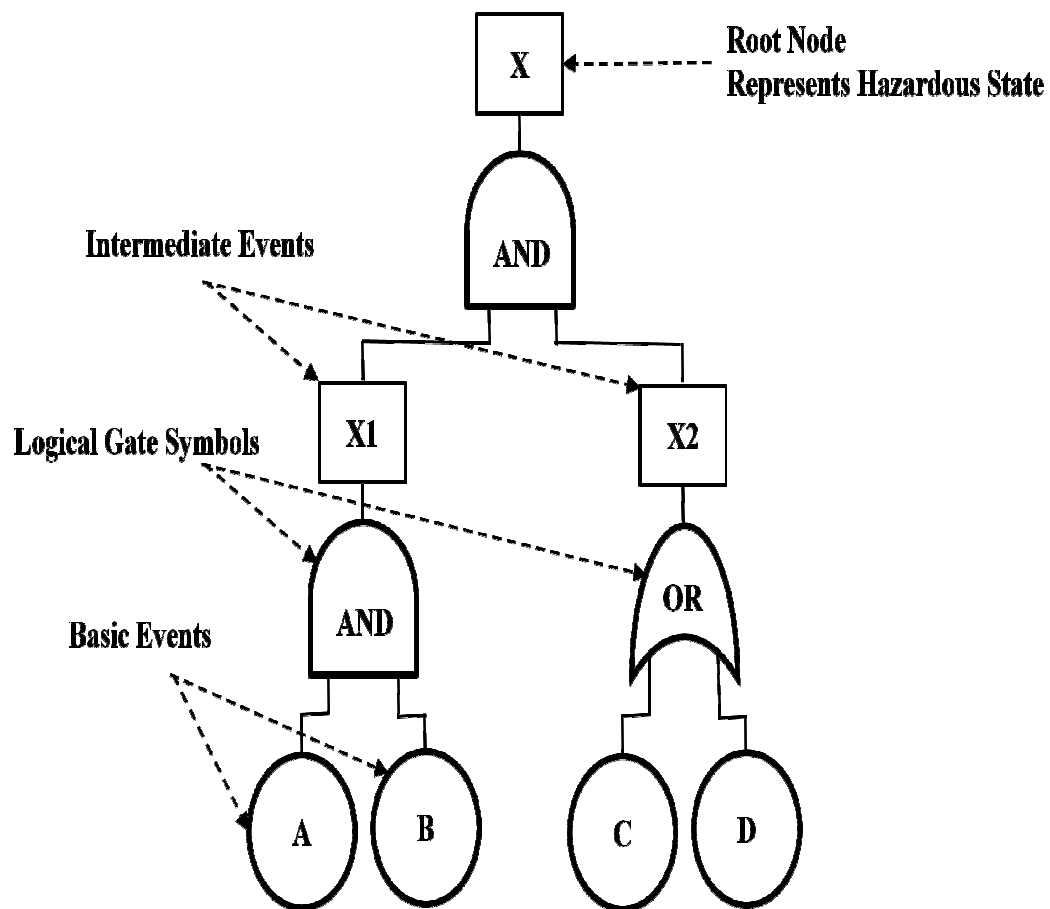
The software fault tree can be analyzed either qualitatively way or quantitatively. These two analysis approaches for software fault trees are explained in following two sections by taking a sample fault tree of Figure 1.1, as an example.

1.3.1 Qualitative Analysis

The objective of qualitative analysis (also known as *cut-set analysis*) is to find out all the possible logical combinations of basic events that can cause the selected hazardous state (root node). A logical combination of basic events leading to hazardous state is known as a *cut set*. A *cut set* is known as a *minimal cut set*, if it contains a minimum number of logically related erroneous events that still can cause the root hazard.

Table 1.1: Symbols used for Fault Tree Events and Logic Gates

Fault Tree Event Symbols		
Events	Symbol	Description
Basic Event		A basic initiating fault (or failure event). A basic event does not need further resolution.
Top Event / Intermediate Event		An event to be analyzed (a root node). It can also be used for intermediate event.
Fault Tree Logic Gate Symbols		
Logic Gates	Symbol	Description
AND		The output event occurs only if all input events occur
OR		The output event occurs if at least one of the input event occurs.

**Figure 1.1 A Sample Fault Tree for Hazardous-state X**

For example, in fault tree as shown in Figure 1.1, the *minimal cut set events* are (A,B,C) and (A,B,D) with Boolean expressions as $A.B+C$ and $A.B+D$ respectively. (Where dot (.) stands for AND operation and (+) stands for OR operation).

1.3.2 Quantitative Analysis

The quantitative analysis of fault tree is used to predict the reliability of the system. The quantitative analysis is used to calculate the probability of the occurrence of root node event using the probabilities of the basic events. For example, in fault tree as shown in Figure 1.1, if the probabilities for the occurrence of basic events 'A', 'B', 'C' and 'D' are 0.2, 0.2, 0.1 and 0.3 respectively, then the probability value for the occurrence of root node 'X' is $(0.2 \times 0.2) \times (0.1 + 0.3)$ i.e. 0.016.

Performing quantitative analysis of any hardware fault tree is easy and feasible because the life expectancy and the expected failure modes of any hardware device/component are generally known in the public domain. However, the failure data about various software related errors is generally not easily available. The quantitative analysis can be used to predict the software reliability of the software system by (i) constructing a generalized software fault tree for a software failure and (ii) using the probabilities of the software related errors. Note that the software reliability is defined as the probability of failure-free operation of a software system for a specified time in a specified environment. If the root node of the software fault tree represents a generalized software failure then the probability of software failure can be computed using the probabilities of the software related errors. The software reliability can be predicted using this computed value of probability of software failure.

1.4 SOFTWARE FAILURE MODES AND EFFECTS ANALYSIS

SFMEA is a forward, inductive and tabular failure assessment approach in the sense that its application starts by first identifying the various failure modes of a software component and then investigating the effects of those failure modes on the whole system. Traditionally, two types of software FMEA approaches have been reported in the literature: *System Software FMEA* and *Detailed Software FMEA* (Goddard, 2000). The system level software FMEA approach covers only the top abstract level functionalities of the system (without any focus on implementation) and can be applied during the early/initial stages of software development such as in the software

requirements analysis and the preliminary design phases. The detailed software FMEA is applied in the later stages of software development when either the pseudo code description or implemented code of various functionalities is available for analysis. Bowles and Wan (Bowles and Wan, 2001) introduced a third SFMEA type named *Interface Software FMEA* to analyze interface related failures that can occur between software and hardware interface modules.

The application of SFMEA approach results in the creation of one or more table known as SFMEA tables. For software systems, there is no universally accepted format or structure for these SFMEA tables. The contents, structure of these tables depend upon the software life-cycle phase in which the approach is applied. A typical SFMEA table generally contains the fields as shown in Table 1.2.

Table1.2: SFMEA Worksheet

Item	Failure Modes	Causes	Effects on the System	Probability of Occurrence
<< Item to be explored for failure analysis>>	<< Failure Mode of the Item>>	<<Causes of the failure mode>>	<< Effects of the failure on the System>>	<< Probability of occurrence of the failure>>

The 'Item' column can have various possible values such as a name of an individual variable (in code level analysis) or name of method/operation of a class (in design phase) etc or name of a use-case related software error (in requirements analysis phase). The focus of the qualitative application of the SFMEA approach is generally to find out the causes responsible for the selected failure modes of an item and to investigate the effects ('Effects' column entries) of these failures on the system. The analysis becomes quantitative in nature, if the probability value for every failure mode of each item is known in advance. Like SFTA, the quantitative application of SFMEA approach is also used to predict the reliability of systems.

1.5 MODELS USED IN UNIFIED MODELING LANGUAGE (UML)

Unified Modeling Language has emerged an industry-modeling standard for effectively representing the static and dynamic aspects of the software system. The UML supports nine types of diagrams that can be used in various phases of software development.

The detailed information about these models can be found in the work of Booch (Booch et al, 2005).

This section gives the overview of the three UML models namely (i) use-cases, (ii) sequence diagrams and (iii) state charts. These three UML models are used in this thesis for developing the automated and semi-automated applications of SFTA and SFMEA approaches. The use-case models and state diagrams are used for the SFTA and SFMEA approaches developed for use in the object-oriented requirements analysis phase whereas the sequence and state diagrams are used for the SFTA and SFMEA approaches developed for the object-oriented design phase.

1.5.1 Use-Case Models

Use-case models (UCM) are introduced in UML as the main requirement analysis tools that are used to depict the abstract level functionalities offered by the system in the form of a graph of *actors* (users of the system) and *use-cases* (functionalities/services offered by the system). A sample UCM is as shown Figure1.2, with two actors ('A' and 'B') and four use-cases (Functionality1, Functionality2, Functionality3 and Functionality4).

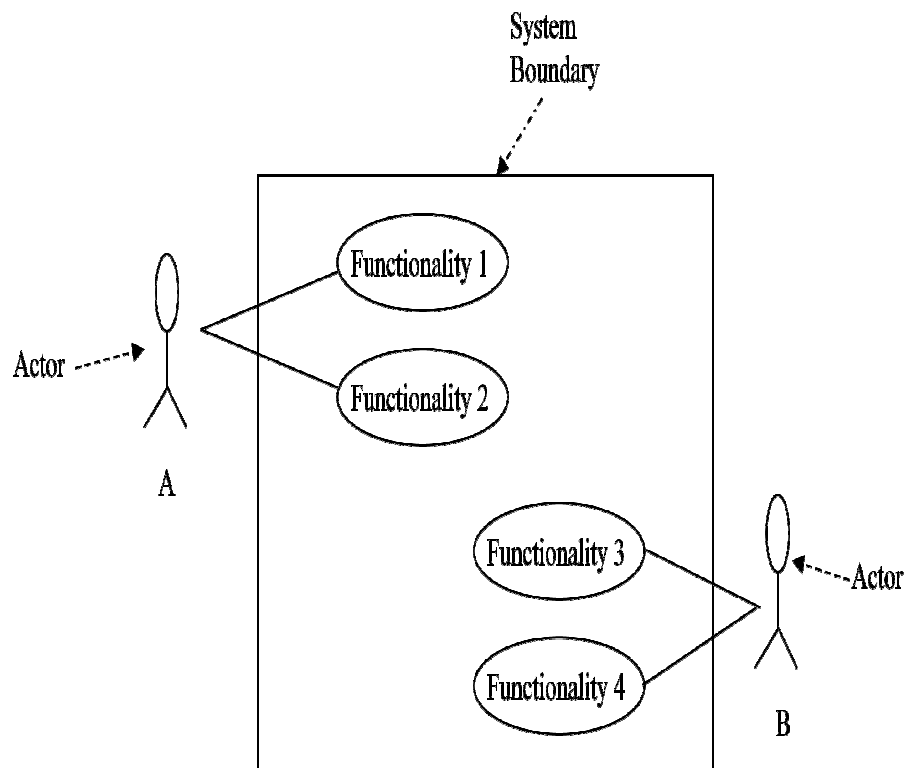


Figure1.2: A Sample Use Case Model

The services or operations initiated by the actor ‘A’ are Functionality1 and Functionality2 whereas the services initiated by actor ‘B’ are Functionality3 and Functionality4.

The formal functional description of each use-case is expressed by filling the details in a use-case realization template (UCRT) with sufficient details in English like natural language. There is no universally accepted structure and format for writing the UCRT, but a typical UCRT contains the items of information as shown in Table 1.3.

Table 1.3: A Sample Use Case Realization Template

Use Case Name	<< Name of the use-case>>
Actor	<<Name of the actor who will initiate the use-case>>
Pre-condition	<<The condition that must be true before initiating the use-case>>
Post-condition	<<The condition that must be true after exiting the use-case>>
Normal Flow of Actions (Main Scenario Action)	
⋮ << Actions to be carried out during normal scenario execution>> ⋮	
Alternative Flow 1	<< Actions to be carried out during alternative scenario-1>>
Alternative Flow 2	<< Actions to be carried out during alternative scenario-2>>
⋮	⋮
Alternative Flow n	<< Actions to be carried out during alternative scenario-n>>

A given use-case functionality may have any number of alternative paths of execution and each such path of execution is known as a *scenario*. Each scenario has its associated list of actions that are executed during the invocation of that scenario.

The whole object-oriented software development process is use-case driven (Jacobson et.al, 1992). The use-cases and the UCRTs developed in the object-oriented requirements phase are used as inputs in the succeeding phases of the object-oriented life cycle.

1.5.2 Sequence Diagrams

Sequence diagrams are the main interaction models in UML which are drawn to show (i) various objects that are participating and collaborating with each other in the execution of a particular use-case scenario and (ii) various messages exchanged among these participating objects. The participating objects are shown on the horizontal dimension whereas the message(s) and the sequences in which they are exchanged are shown on the vertical dimension. The participating objects can communicate with each other by sending either synchronous or asynchronous type of message. The symbols along with their meaning, which are used to draw the sequence diagram, are shown via sample sequence diagram of Figure 1.3.

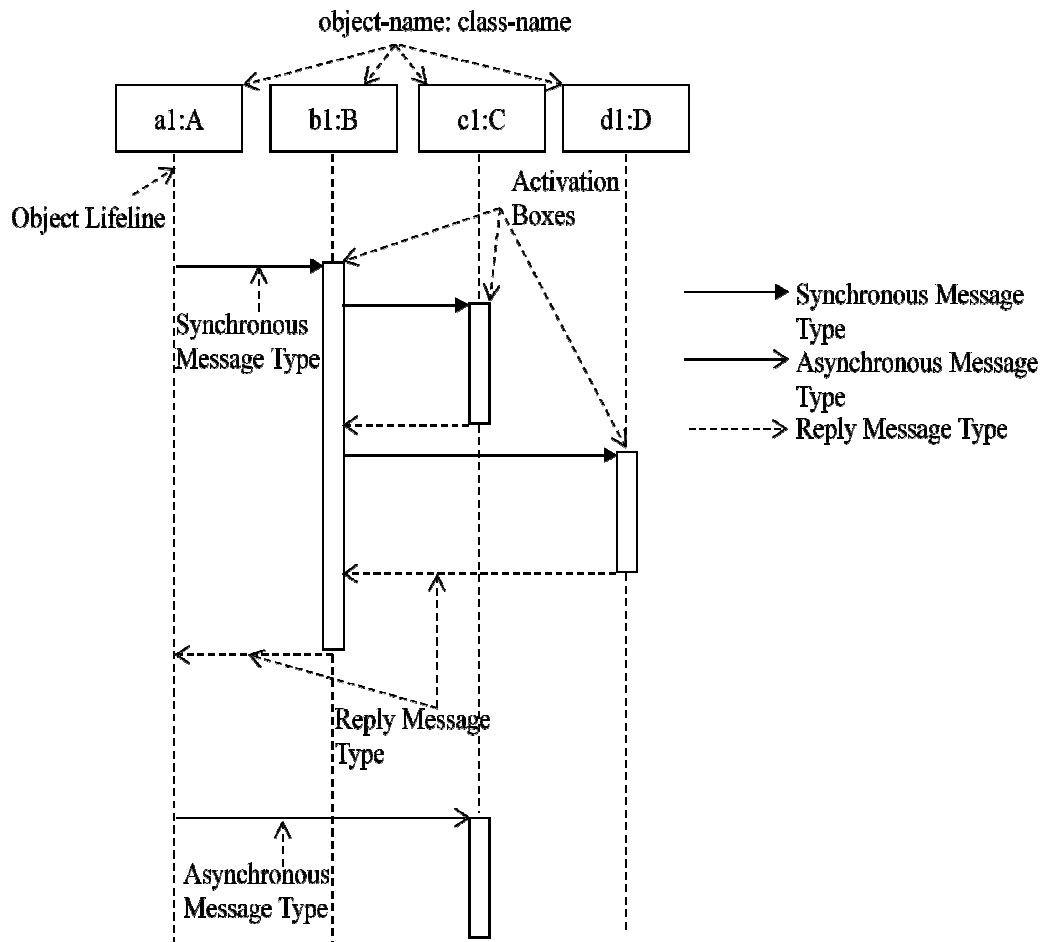


Figure 1.3: A Sample Sequence Diagram

When a sender object sends a synchronous message to any other receiver object, then the sender object is blocked from further communication with any other object unless it does

not get the reply of the previously sent message whereas in case of asynchronous message the sender object is not blocked. How much time an object will take to respond to a received message, is indicated via an activation box (rectangle type symbol) on the lifeline of the receiver object.

Software developers generally use various types of stereotypes such as <<controller>>, <<interface>>, <<device>>, <<controller>> along with object names to represent the type of the object participating in the interaction. The <<controller>> stereotype is used for a software controller object. The <<device>> stereotype is used for a hardware or device type object. Software controller type objects manage the working/functionality of the device type of objects. The <<interface>> stereotype is used for a interface type object which acts as a communication medium between a software controller and a hardware/device types of objects. Apart from these three stereotypes, the developers can introduce any other stereotype also for any other type of object depending upon the requirements. Depending upon the requirement, some interaction operators are also used in the sequence diagram and five of these interaction operators are explained below.

(i) The 'alt' operator

The 'alt' word stands for alternative set of messages. In Figure 1.4, the message 'Message 1' is sent only if condition $x=0$ is true otherwise message 'Message 2' is sent. The 'alt' represents a 'if then else' type construct. A sequence diagram can have any number of 'alt' block and these 'alt' blocks can be in nested form too.

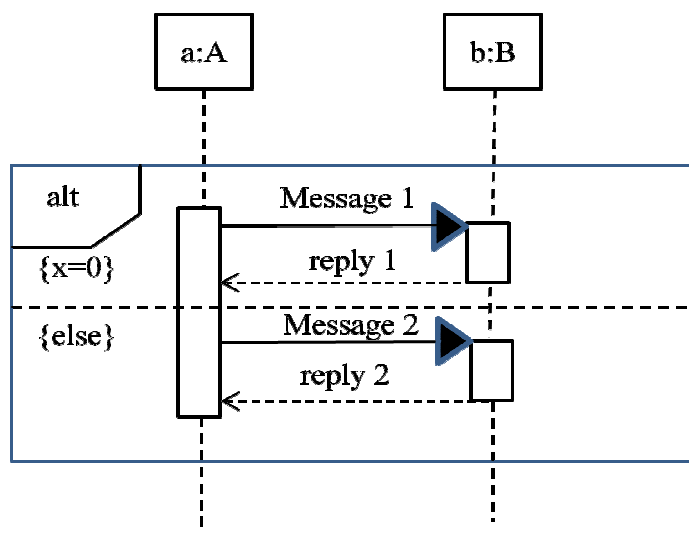


Figure 1.4: Sequence Diagram Example representing an 'alt' block

(ii) The 'opt' operator

The 'opt' operator stands for optional message block. The messages in the block are sent only if the condition associated with the block is satisfied otherwise the messages in the block are skipped. An example of 'opt' block is shown in the sequence diagram of Figure 1.5. The 'opt' represents a 'if then' type construct without an else option.

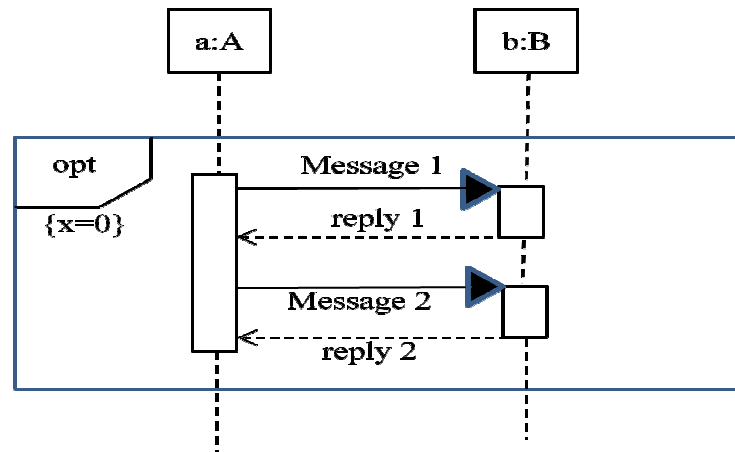


Figure 1.5: Sequence Diagram with an 'opt' block

(iii) The 'break' operator

The 'break' operator is similar to 'opt' block but represents an exceptional scenario where either the messages from the 'break' block are sent or the messages after the break block are sent. A sequence diagram with a 'break' block is shown in Figure 1.6.

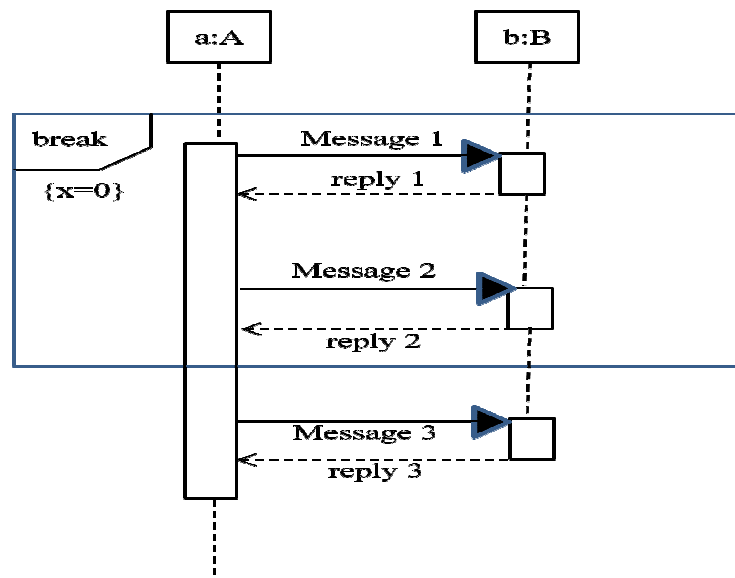


Figure 1.6: Sequence Diagram with a 'break' block

If the condition 'x=0' is satisfied then only the messages in the 'break' block i.e. 'Message 1' and 'Message 2' are sent and after that the execution stops (i.e. the message 'Message 3' which is after the 'break' block is skipped). However, if the condition is false then the messages in the 'break' block are skipped (not sent) and the messages outside the block i.e. message 'Message 3' is sent only.

(iv) The 'loop' operator

The 'loop' interaction operator is used to repeat message sequence either for a fixed number of times or until a condition is satisfied. A sequence diagram with a 'loop' interaction operator is shown in Figure 1.7 and according to this figure the messages 'Message 1' and 'Message 2' are sent 5 times.

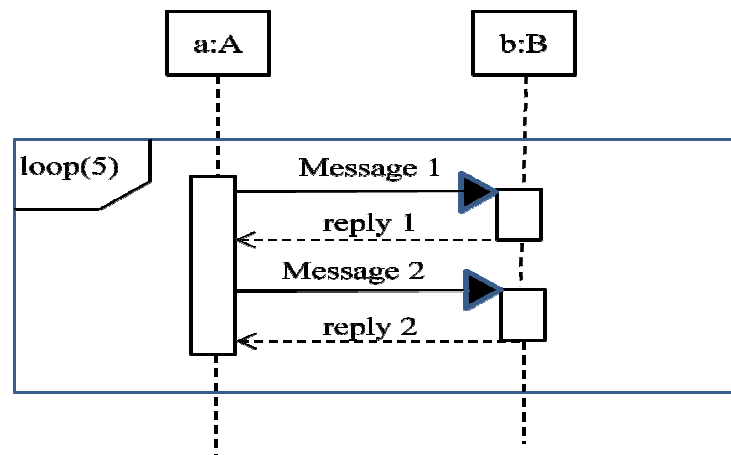


Figure 1.7: Sequence Diagram with a 'loop' block

(v) The 'par' operator

The 'par' operator stands for parallel messages. The messages in the 'par' block are sent in parallel. The send events of the messages that are sent in parallel can be interleaved. A sequence diagram with 'par' block is shown in Figure 1.8.

1.5.3 State Charts

State diagram is also a dynamic interaction model of UML used to depict the various states an object that it transits in its lifetime in response to an outside stimulus or event. Some of the symbols used to draw the state diagrams are shown in Figure 1.9. An object can make the transition from one state to another, either by sending a message to some other object or by receiving a message from some other object.

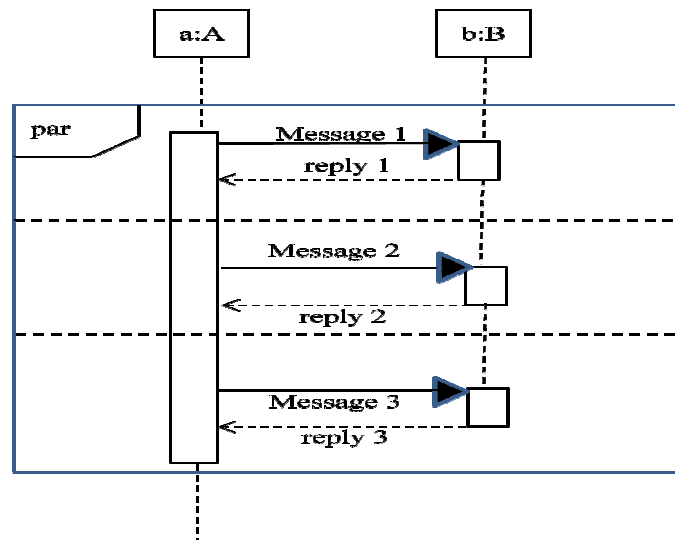


Figure 1.8: Sequence Diagram with a ‘par’ block

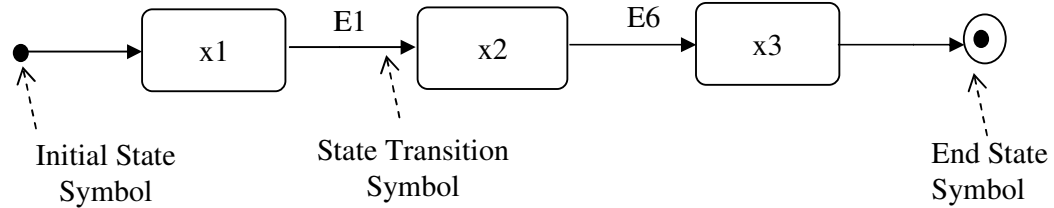


Figure 1.9: A Simple UML State Diagram

The objects participating in an interaction can experience either a simple state transitions as shown in Figure 1.9 or concurrent sub-state transitions as shown in Figure 1.10. In Figure 1.10, the object experience concurrent state transitions in states ‘x2’ and ‘x4’ on the occurrence of the same event ‘M2’.

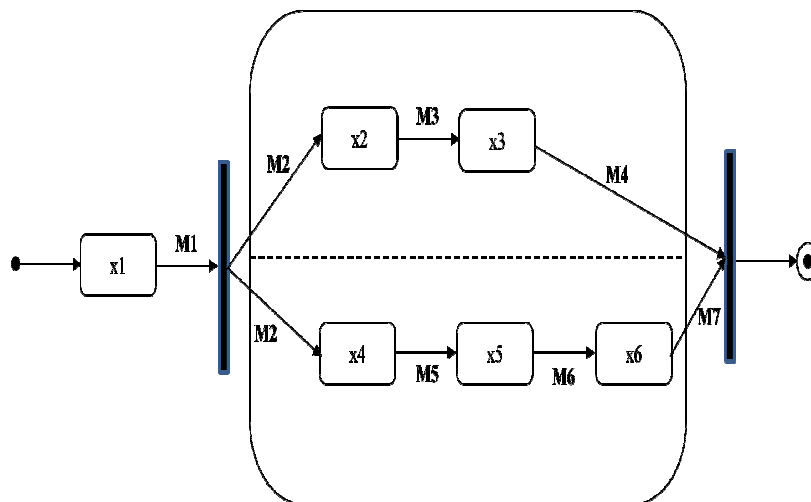


Figure 1.10: UML State Diagram with Concurrent Sub-states

This chapter presents the critical review of the published work about the applications of SFTA and SFMEA approaches in two phases, namely requirements analysis and software design, of software development process.

2.1 APPLICATIONS OF SFTA APPROACH

2.1.1 In Software Requirements Analysis Phase

SFTA application approach for requirements analysis is used either to identify the safety related faults in the given requirement specification or to elicit the required safety requirements needed to mitigate the considered hazardous state by fault tree construction.

Mojdehbakhsh et al (1994) described a four step approach to identify the safety faults in the software requirements specified using the Statemate Case tool (Harel et al., 1990). The first step of the approach constructs a software requirement, fault tree (SRFT) automatically from the specified requirements and identifies the safety faults. The second step verifies and validates the SRFT constructed in the first step. The requirements are generated in the third step, which are verified and validated in the fourth step.

Melhart (1995) used an augmented form state model known as an external interaction model (EIM) (Melhart, 1990) for specifying the software requirements and provided templates to construct software fault trees directly from an input EIM. The results of the analysis can be used for requirements modification and correction.

Ratan et al (1996) developed a fault tree generator tool to generate fault trees automatically from the requirements specified in state-based requirements specification language known as Requirements State Machine Language (RSML) but the tool generates a fault tree only for one-step backward at a time. However, the main strength of the tool is that it also checks for the consistency and completeness of the specified requirements during fault tree construction.

Gorski and Wardzinski (1996) presented a four step manual approach to derive the safety requirements using fault trees defined in a formal model named Common Safety Description Model (CSDM). In the first step a fault tree is constructed manually by an analyst using domain expertise. The second step defines the constructed fault tree in a formal way using CSDM. The third step computes the minimal cut-set events, which are used to derive the necessary requirements in the fourth step.

Tsuchiya et al (1997) proposed an FTA-based technique to derive the necessary safety features from the requirement specifications expressed in English like natural language for object-oriented systems.

Lutz and Woodhouse (1997) integrated the application of the SFTA approach with a SFMEA approach to analyze the requirements of two spacecraft systems named Cassini and Galileo. The integrated application of SFTA and SFMEA approaches enhances the consistency, completeness and robustness of the derived safety features.

Hansen et al (1998) used the features of fault trees and the duration calculus approaches to derive the necessary software safety requirements. Helmer et al (2002) has used SFTA approach to model the intrusions to determine and verify the security requirements for an intrusion detection system (IDS).

The works reported by Dehlinger & Lutz (2004) and Feng & Lutz (2005), used the applications of SFTA and SFMEA approaches for the safety analysis of the product-line requirements. The product-line requirements have been specified using a technique known as Commonality and Variability Analysis (CVA). CVA is a well-known approach used especially in product-line software engineering for identifying the mandatory requirements (commonalties) or optional requirements (Variabilities or Variations) for a particular product-line member. However, the addition of new variation poses a variation management difficulty because they may introduce new dependencies, which make it difficult to provide assurance for safety. To overcome this problem, Liu et al (2007) introduced a SFTA assisted technique to perform safety analysis on the variations in a product line using state-based modeling.

Lutz et al (2007) used SFTA and SFMEA approaches during the application of '*Obstacle Identification*' step of an approach named *obstacle analysis* to identify the contingency requirements for an unpiloted aerial vehicle (UAV).

Balz and Goll (2005) introduced the FTA technique in use-case based system development process. The approach is manual in nature and is applied to each use-case separately. Use cases (Jacobson et al, 1992; Cockburn, 2000) have been introduced in UML based software development process as the standard requirements analysis tool. Douglass's (2009) work stressed upon first drawing the fault tree using the domain expertise and then linking the leaf nodes (basic events) of the constructed fault tree with the respective use-case functionalities.

Gupta et al (Gupta et al, 2012) presented an eight-step approach for the integrated applications of SFMEA and SFTA approaches in the textual description of the given use case functionality.

The work reported by Tiwari et al (Tiwari et al, 2012) also presented an integrated SFTA and SFMEA application approach that takes formal use-case description as the primary input. However, the integrated approach first converts the formal use-case specification into a tree known as *software success tree* (SST) and then constructs a fault tree by complementing the nodes of SST. The results of SFTA approach have been further analyzed using SFMEA approach.

Summary of Applications

To identify safety related faults/errors, SFTA approach has been applied mostly to the requirements specified using any state-based representation, either by using a tool (Mojdehbakhsh et al, 1994) or a model (Melhart, 95) or a language (Rattan et al, 1996). In some published works, the safety requirements are derived by constructing fault trees by taking some form of requirements specifications as the primary inputs where input requirements are specified either using English like natural language (Tsuchiya et al, 1997; Lutz and Woodhouse, 1997) or use-cases (Balz and Goll, 2005; Douglass, 2009; Gupta et al, 2012) or CVA approach (for product-line requirements) (Dehlinger and Lutz, 2004; Feng and Lutz, 2005 and Liu et al, 2007). In some cases, safety requirements are derived directly by constructing a fault tree without taking any requirement specification as an input (Gorski and Wardzinski, 1996; Hansen et al, 1998, Helmer et al, 2002). A comparative summary of the SFTA applications at requirements analysis phase is shown in Table 2.1.

Table 2.1: Summary of SFTA Applications at Requirements Analysis Phase

Authors	Requirements Representation	Approaches Used	Application Method (Manual/Automated)	Objective
Mojdehbakhsh et al (1994)	STATEMATE - a state based tool	SFTA	Automated	To find the safety faults in the given requirements
Melhart (1995)	EIM (External Interaction Model- a state based model)	SFTA	Template-Based/ Manual	To analyze the specified requirements
Ratan et al (1996)	RSML (-state based language)	SFTA	Partially Automated	To check the consistency and completeness of requirements
Gorski & Wardzinski (1996)	-	SFTA	Manual	To derive the real time requirements
Tsuchiya et al (1997)	Natural Language (English)	SFTA	Manual	To derive the safety requirements
Lutz & Woodhouse (1997)	Natural Language (English)	SFMEA + SFTA	Manual	To identify the ambiguity, inconsistency and missing requirements
Hansen et al (1998)	-	SFTA + Duration Calculus	Manual	To derive the safety requirements
Helmer et al (2002)	-	SFTA	Manual	To derive the security requirements
Balz & Goll (2005)	Use-cases	SFTA	Manual	To derive the safety features for a selected use-case functionality
Dehlinger & Lutz (2004) and Feng & Lutz (2005)	Commonality and Variability Analysis (CVA)	SFTA + SFMEA	Manual	To identify the missing and new safety requirements in product-line requirements
Liu et al (2007)	Commonality and Variability Analysis (CVA)	SFTA + SFMEA	Manual	To perform the safety analysis of software product lines using state-based modeling and SFTA approaches
Lutz et al (2007)	-	Obstacle Analysis + SFTA + SFMEA	Manual	To aid the application of the Obstacle Analysis approach to derive the contingency requirements
Douglass (2009)	Use-cases	SFTA	Manual	To perform the safety analysis of the given use-case functionality
Gupta et al. (2012) & Tiwari et al (2012)	Textual use-case description	SFTA + SFMEA	Manual	To analyze the use-case based requirements

2.1.2 In Software Design Phase

UML has emerged as a de facto industry-modeling standard for modeling the static as well as dynamic aspects of software systems. That is why, the SFTA applications at software design phase are applied mostly on UML models. The objectives of applying SFTA approach at software design phase are to identify the flaws in the design of a software module/component and to identify the modules that are critical from a safety point of view.

Pai and Dugan (2002) presented an algorithm to automatically synthesize dynamic fault trees (DFTs) from UML system models for reliability analysis. DFTs are extensions of static fault trees, especially to model fault tolerant features such as redundancy etc. The reliability related information has been embedded in the UML models. It is to be noted that Pai and Dugan (2002) have used UML models for modeling certain hardware features such as redundancy, spares and reconfiguration.

Towhidnejad et al (2002, 2003) provided a partial paradigm in the form of guidelines for converting UML activity, state and sequence charts/diagrams to fault trees. The approach is manual and time-consuming.

Hawkins and McDermid (2002) and Hawkins (2006) used UML collaboration diagrams to identify unsafe elements in the form objects and also to construct fault trees. The unsafe behaviors of the selected unsafe elements are further identified from state charts using a technique known as Function Failure Analysis (FFA).

Lu et al (2005) embedded the fault tree related information in UML component-based UML model (CBUM) so that hazard analysis and safety analysis can be performed at the same time. Lu, Halang and Zalewski (2005) embedded the information about the elements of two approaches named Hazard and Operability (HazOp) and Fault Tree Analyses (FTA) into UML component models.

Kim et al (2010) presented rules and algorithms to automatically transform a hazard represented by fault trees to state machine diagram. The objective is to bridge the gap between the desired behavior (represented in the form of an original state machine diagram) and undesired behavior of the system (represented via fault trees).

Lauer and German (2011) presented an approach to automatically synthesize fault tree from UML component architecture model for reliability analysis during the design stage.

The UML models used in the approach as such do not depict any functional aspect of the system. Rather, UML has been used to model certain reliability related attributes such as fault propagation and fault containment. The objective of the work is reliability analysis not safety analysis.

Table 2.2: Summary of SFTA Applications at Design Phase

Authors	Application Method (Manual/Automated)	Purpose (Safety Analysis/Reliability Analysis)	Objective
Pai & Dugan (2002)	Automated	Reliability Analysis	Presented an algorithm to construct dynamic fault tree from UML models
Towhidnejad et al (2002,2003)	Manual	Safety Analysis	Provided guidelines to construct software fault trees from UML activity, sequence and state diagrams
Hawkins (2006) Hawkins and McDermid (2002)	Manual	Safety Analysis	Recommended to construct fault tree from collaboration diagram and integrated the application of Fault Trees and Functional Failure Analysis (FFA) approaches
Lu et al (2005)	Manual	Safety Analysis	Embedded the fault tree related information in UML component-based UML model (CBUM)
Lu, Halang & Zalewski (2005)	Manual	Safety Analysis	Embedded the information about the elements of two approaches named Hazard and Operability (HazOp) and Fault Tree Analyses (FTA) into UML component models
Kim et al (2010)	Manual	Safety Analysis	Presented rules and algorithms to automatically transform fault trees to state diagrams
Lauer & German(2011)	Automated	Reliability Analysis	Presented an approach to automatically synthesize fault tree from UML component architecture model for reliability analysis during the design stage

Summary of Applications

SFTA applications on UML models are applied either for the reliability analysis of system or for the safety analysis. For reliability analysis purpose, the approach is automated and the reliability related information is embedded in the models itself. For the safety analysis purpose, the application of the approach is either manual or semi-automated. A summary of applications of the SFTA approach on the design phase is given in Table 2.2.

2.2 APPLICATIONS OF SFMEA APPROACH

2.2.1 In Software Requirements Analysis Phase

Lutz and Woodhouse (1996) applied SFMEA approach to the requirements analysis of critical spacecraft software and the application was found to be successful not only in identifying the ambiguities and inconsistencies in the specified requirements but also to identify various missing requirements.

Lutz and Woodhouse (1997) described a two step approach named Bi-Directional Safety Analysis (BDSA) that integrates the applications of SFMEA and SFTA approaches. The SFMEA approach is applied to the specified requirements in the first step and SFTA has been applied in the next step on the results obtained in the first step. The requirements are expressed in English text.

The same BDSA approach was extended for analyzing the product-line requirements (Lutz, 1998; Feng and Lutz, 2005) with the two differences. Firstly, the requirements have been specified using a CVA approach (for product-line members). Secondly, both approaches are applied separately to the requirements and the results of both approaches are compared at the end to find out any mismatch.

Wentao and Hong (2009) presented a manual SFMEA approach that can be used to use-case based requirements analysis phase and demonstrated its merits by applying it to the use-case model of a typical bank Automated Teller Machine (ATM). The input for the approach is the formal description of a given use-case functionality.

Troubitsyna (2011) used manual SFMEA approach to augment a given use-case model with a fault tolerance mechanism. She has recommended creating and defining an auxiliary use case (for each actual use-case) to model error recovery. The approach has been applied to a use case model of an autonomous robot.

Nggada's work (2012) demonstrated the manual application of the SFMEA approach on the use-case model of a Brake by Wire (BBS) System. The works reported by Gupta et al (2012) and Tiwari et al (2012) demonstrates the integrated use of SFTA and SFMEA approaches in use-cases which have been discussed in Section 4.1 also.

Summary of Applications

There exists very little literature work about the application of SFMEA approaches for requirements analysis. The SFMEA approach has been used mostly in conjunction with SFTA approach for requirements analysis. In some cases (where SFTA and SFMEA are used together) the results of SFMEA approach have been used as an input for the application of the SFTA approach (Lutz and Woodhouse 1996, 1997; Gupta et al, 2012). In some cases the results of SFTA approach have been used as an input for the application of SFMEA approaches (Tiwari et al, 2012). A summary of SFMEA applications either alone or in conjunction with SFTA approach is shown in Table 2.3.

Table 2.3: Summary of SFMEA Applications at Requirements Analysis Phase

Authors	Requirements Representation	Approaches Used	Application Method (Manual/Automated)	Objective
Lutz & Woodhouse (1996,1997)	English Text	SFMEA + SFTA	Manual	To identify ambiguity, inconsistency in the requirements as well as to identify missing requirements
Lutz (1998) and Feng & Lutz (2005)	Commonality and Variability Analysis (CVA)	SFMEA + SFTA	Manual	To identify the missing and new safety requirements in product-line requirements
Wentao & Hong (2009)	Textual Description of Use Case	SFMEA	Manual	Safety Analysis of Use Case Models
Troubitsyna (2011)	Use Case Model	SFMEA	Manual	Augmenting Use case Model with Fault Tolerant Features
Nggada(2012)	Use Case Model	SFMEA	Manual	To perform the failure analysis at use-case based requirements analysis phase
Gupta et al. (2012) & Tiwari et al (2012)	Textual use-case description	SFTA + SFMEA	Manual	To analyze the use-case based requirements

2.2.2 In Software Design Phase

Guiochet and Baron (2003) identified eleven types of message error models for the application of a FMECA technique on UML sequence diagrams and used these error models for the risk analysis of medical robot (Guiochet and Baron, 2004).

Hecht and Hecht (Hecht and Hecht, 2004) described a computer-aided SFMEA approach that can be used in two stages of software development: concept phase and design/implementation phase. The use case diagram can be a potential input for the application of the concept phase SFMEA approach, whereas for design/implementation phase SFMEA approach, the inputs can be the methods/operations (equivalent to a subroutine of assembly languages) of various classes.

The work presented by Ozarin (Ozarin, 2004) stressed upon the application of the Software FMEA approach during the whole software design phase by taking various UML diagrams and software grouping constructs as primary inputs. The information about which UML diagram should be taken as a possible input, for the application of SFMEA approaches at various stages of a software development, can be found in the works of Ozarin (Ozarin, 2004).

Hassan et al (Hassan et al, 2005) proposed a five step UML based severity assessment methodology based upon the integrated applications of three approaches named Functional Failure Analysis (FFA), FMEA and FTA. The FFA is applied during the first step by taking use-case diagram and system scenario diagrams as potential inputs. The FMEA approach is applied in the second step by taking scenario sequence diagrams and component interaction diagrams as inputs. The third step applies an FTA approach on the outputs of the first two steps.

David et al (David et al, 2008) described an approach for generating an FMEA table from a sequence diagram, but requires that a database of dysfunctional behaviors of various classes (involved in the sequence diagram) should be known in advance. The dysfunctional behavior database for various classes provides the basis for identifying the failure modes for each class.

Summary of Applications

The objective of applying SFMEA during the UML design stage is to do the failure analysis of the given UML model. The application process is manual, labor-intensive and

costly. We are not able to trace any research paper describing the automation of the approach for any UML model. A summary of the published work about SFMEA applications on the UML models is given in Table 2.4.

Table 2.4: Summary of SFMEA Applications at Software Design Phase

Authors	Application Method (Manual/Automated)	Objective
Guiochet and Baron (2003,2004)	Manual	Identified Eleven types of message errors and used them for the risk analysis of a medical robot
Hecht and Hecht (2004)	Manual	Introduced Concept Phase SFMEA and Design/Implementation Phase SFMEA
Ozarin(2004)	Manual	Gives the information about which UML model should be used as input for the application of the SFMEA approach during various phases of software development
Hassan et al (2005)	Manual	Integrates three approaches named FFA,FMEA & FTA for the severity assessment of UML models
David et al (2008)	Manual	Described an approach to generate an FMEA table from sequence diagrams

2.3 RESEARCH GAPS

Since the inception of both these approaches in the software domain, there are ongoing efforts in the research community to automate or semi-automate the applications these approaches in almost every phase of software development process. These efforts are found to be successful to some extent, so far, only at coding phase (Friedman, 1993; Reid, 1994; Winter and Shimeall, 1995; Snooke and Price, 2011).

As per the analysis given in Table 2.1, the application of SFTA has been applied broadly to four categories of requirements specifications as (i) requirements specified using state-based representation, (ii) requirements specified in English like natural language, (iii) requirements specified using CVA approach for product-line engineering and (iv) requirements specified using use-cases. So far, SFTA application has been automated and semi-automated only for the requirements specified using state-based representation. Whereas, for other types of requirements specifications, especially for the requirements specified using use-cases, the application of SFTA is still manual.

Similarly, as per the analysis shown in Table 2.2, the application of SFTA has been automated for reliability analysis but not for safety analysis. The application of SFMEA approach is also still manual in both requirements analysis and software design phases.

Based upon the analytical review of the applications of both SFTA and SFMEA approaches in the requirements analysis and software phases, the following research gaps have been identified.

Research Gap I: In use-case based requirements analysis process the application of both SFTA and SFMEA approaches is still manual.

Research Gap II: The application of both SFTA and SFMEA approaches in UML based software design phase is still manual from safety analysis point of view.

Software Fault Tree Analysis Approach in Use-Case based Requirements Analysis Phase

This chapter presents the automated SFTA technique in use-case based requirements analysis phase. The application of the SFTA approach is automatic where a fault tree for a given hazardous-state for a safety-critical system is constructed automatically. The formal textual description of a given use-case functionality, the UML state diagrams of the participating components and the hazardous-state (for which a fault tree is to be constructed) of the system are required as inputs in the proposed SFTA approach. The proposed technique has been validated by applying it to the use-case functionalities of two safety-critical applications, namely Elevator Control System (ECS) (Gomaa, 2000) and Rail Track-Door Control System (RTCS) (Medikonda and Ramaiah, 2010).

3.1 PURPOSE OF THE PROPOSED SFTA APPROACH

The use-case based FTA approach introduced by Balz and Goll (Balz and Goll, 2005) is manual in nature and fault trees are constructed manually for the selected use-case functionalities. Douglass (Douglass, 2009) stressed upon first constructing fault trees using domain expertise and then linking the nodes of the fault tree with respective use-cases. The works reported by Gupta (Gupta et al, 2012) and Tiwari (Tiwari et al, 2012) applied SFTA to a textual description of the selected use-case functionality. In all these cases, the fault trees were constructed manually either to elicit the safety requirements or to verify the already derived safety requirements and the application of the SFTA approach in use-case based requirements analysis phase is manual and time-consuming. The objective of the proposed approach is to integrate and automate the application of the SFTA approach in use-case based requirements analysis process so that the necessary safety requirements are derived right in the requirements analysis phase. The approach does a backward analysis to identify the errors responsible for the occurrence of the selected hazardous state.

3.2 ASSUMPTIONS FOR THE PROPOSED SFTA APPROACH

The following two assumptions are made for the proposed approach and a brief explanation for each assumption is given in the following sections.

(i) The given use-case functionality and the UML state diagrams for various participating components are complete and correct.

The approach operates with the assumption that the supplied formal functional description of the selected use-case functionality is correct and complete. The completeness of the use-case functionality means that *'no event has been missed-out in the description'*. The correctness of the use-case functionality means *'the sequence in which the various events of the selected use-case functionality will execute, are specified correctly'*. In the same way, it is also assumed that the supplied UML state diagrams are correct (*state transition events have been correctly specified*) and complete (*no state for any component has been missed-out*). Any error either in the use-case description file or in the states of the participating components affects the correctness and completeness of the constructed software fault trees.

(ii) No participating component experience concurrent state transitions

The approach also operates with the assumption that corresponding to any state transition event (an event where any component is changing its state) there is a single state transition experienced by the component. The approach first maps the events of the selected use-case functionality against the states of the participating components and then records the state transition errors corresponding to each state transition event. If a component experiences concurrent state transitions (means minimum two state transitions corresponding to a single state transition event) then the recording of state transition errors will be a challenging if not impossible task. The presented approach does not handle this situation.

3.3 HAZARDOUS-STATE DEFINITION

In the proposed SFTA approach, the hazardous-state of the system is expressed using the states of the participating components, either in *atomic form*, (a hazardous-state involving the state of a single component) or in *composite form* (a hazardous-state involving the states of more than one component).

A hazardous-state in atomic form is used to indicate the situation where a participating component fails to make its expected/desired state transition and is represented using a negation and equality symbols as '!='. The general syntax for an atomic hazardous-state is ' $c \neq s$ ' where ' c ' is the name of a component and ' s ' is one of the state(s) of ' c '.

The composite hazardous-state can involve states from multiple components and can use both negation ('!=') and true (=) type symbols. The use of negation ('!=') symbol indicates that a component has failed to change its state, whereas a true (=) symbol indicates that a component has successfully changed its state. The states from multiple components are joined by an 'AND' operator.

Some examples of both atomic and composite hazardous-states are illustrated in Table 3.1. These hazardous-states are selected from two safety-critical applications mentioned in the beginning of this chapter. The participating components in the ECS application are Door (with valid states as 'opened' or 'closed') and Motor (with valid states as 'moving' or 'stopped'). Whereas, in the RTCS application the participating components are Track_Door (with valid states as 'opened' or 'closed') and Track_Signal (with valid states as 'red' or 'green').

Table 3.1: Hazardous-State Examples

Hazardous-State	State	Description
Door != closed	Atomic	The door has not closed (ECS Application)
Motor != moving	Atomic	The motor has not moved (ECS Application)
Door != closed AND Motor = moving	Composite	The door has not closed, but Motor has moved (ECS Application)
Track_Door != closed AND Signal = green	Composite	The Track_Door is not closed but Signal has gone green (RTCS Application)
Door = opened AND Motor = moving or Motor = moving AND Door = opened	Composite	At any point of time, the system should not have both Door and Motor components are in 'opened' and 'moving' states respectively (ECS Application)
Track_Door = opened AND Track_Signal = green or Track_Signal = green AND Track_Door = opened	Composite	At any point of time, the system should not have both Track-door and Signal components are in 'opened' and 'green' states respectively (RTCS Application)

The composite hazardous-state can be used to represent two different types of hazardous situations and the detailed information about the syntax and validity conditions of both these types of the composite hazardous-state are discussed in more detail in Section 3.5.

3.4 OVERVIEW OF THE PROPOSED SFTA APPROACH

The proposed SFTA approach is a five step approach and takes the formal use-case description file and the UML state diagrams of the participating components as inputs. The UML state diagrams are accepted as inputs in machine readable form i.e. XMI (XML Metadata Interchange) format. The Altova UML (Altova-UModel, 2014) is used to draw the required state diagrams and each one of them is exported to XMI format using the same tool.

The description of a given use-case functionality can have any number of uniquely executable paths known as *scenarios* and each such *scenario* is represented by a unique *event-sequence*. The ‘*Event-Sequence*’ of a particular scenario gives the information about the sequential order in which the events of that scenario will execute. In the first step, the objective is to extract the *event-sequence* for each possible scenario by taking the use-case description file as an input. The successful execution of the first step results in the instantiation of one or more instances of ‘*Event-Sequence*’. The extracted event sequences are saved in a tabular structure. The definition of various fields/columns of this are shown Table 3.2. Each such ‘*Event-Sequence*’ table created represents the event sequence of a particular scenario of the given use-case functionality.

Table 3.2: Structure of Event-Sequence Table

Event#	Precondition	Event-Name	Logical-Time
<<A Unique Event Number Assigned to each executable event>>	<< Precondition that must be true before the execution of the event>>	<< Name of the event as used in use-case description file>>	<< An integer value that represents the sequence number of the event in the event sequence>>

The second step determines the correct functional state of the system by mapping the events of various scenarios against the states of the participating components. The state diagrams for the participant components are drawn in the second step and each state diagram is exported to XMI format using the above mentioned Altova UML tool. The ‘*Event-Sequences*’ and the state diagram XMI files are used as inputs in this step. The application of the second step results in the creation of one or more

instances of a table named ‘*Event-Sequence-State-Transition*’ and all these tables collectively represent the correct functional state of the system. The structure of the ‘*Event-Sequence-State-Transition*’ depends upon the number of components for which state diagrams are supplied as inputs. If the state diagrams are drawn for two components, namely ‘X’ and ‘Y’ then the structure of the ‘*Event-Sequence-State-Transition*’ will have three fields as shown in Table 3.3.

Table 3.3: Structure of Event-Sequence-State-Transition

Event#	X	Y
<<A Unique Event Number Assigned to each executable event>>	<<State of X component during the execution of the event>>	<<State of Y component during the execution of the event>>

In the third step, the erroneous states of the system are identified. The state transition events (events where any component is making a state transition) are filtered from various ‘*Event-Sequence-State-Transition*’ tables and then the state related errors (the errors which prevent the component from making its desired/required state transition) are recorded against the filtered state transition events. This information is stored in a table named ‘*State-Transition-Error*’. The structure of this table has four fields as shown in Table 3.4.

Table 3.4: Structure of State-Transition-Error Table

Error#	Error-Name	Event#	Effect
<<A Unique Error Number Assigned to each Error>>	<<Description of Each Error>>	<< Event# where the error has occurred>>	<< Effect of the Error>>

The fourth step generates a software fault tree for the specified hazardous-state of the system. The ‘*Event-Sequences*’ (extracted in the first step), ‘*Event-Sequence-State-Transitions*’ (determined in the second step) and a ‘*State-Transition-Errors*’ (identified in the third step) are used as inputs. Recall that the ‘*Event-Sequences*’ and ‘*Event-Sequence-State-Transitions*’ collectively represent the correct functional state of the system, whereas the ‘*State-Transition-Error*’ represents the erroneous state of the system. The fourth step generates one or more XML files and each such XML file represents a software fault tree for a particular hazardous-state. The graphical fault tree is constructed by using these XML files as an input to the fault tree creator and analysis tool named ‘FaultCAT’ (FaultCAT, 2003) in the last step.

The overview of the first four steps of the proposed SFTA algorithm is shown in Figure 3.3 and the SFTA algorithm is explained in the next section.

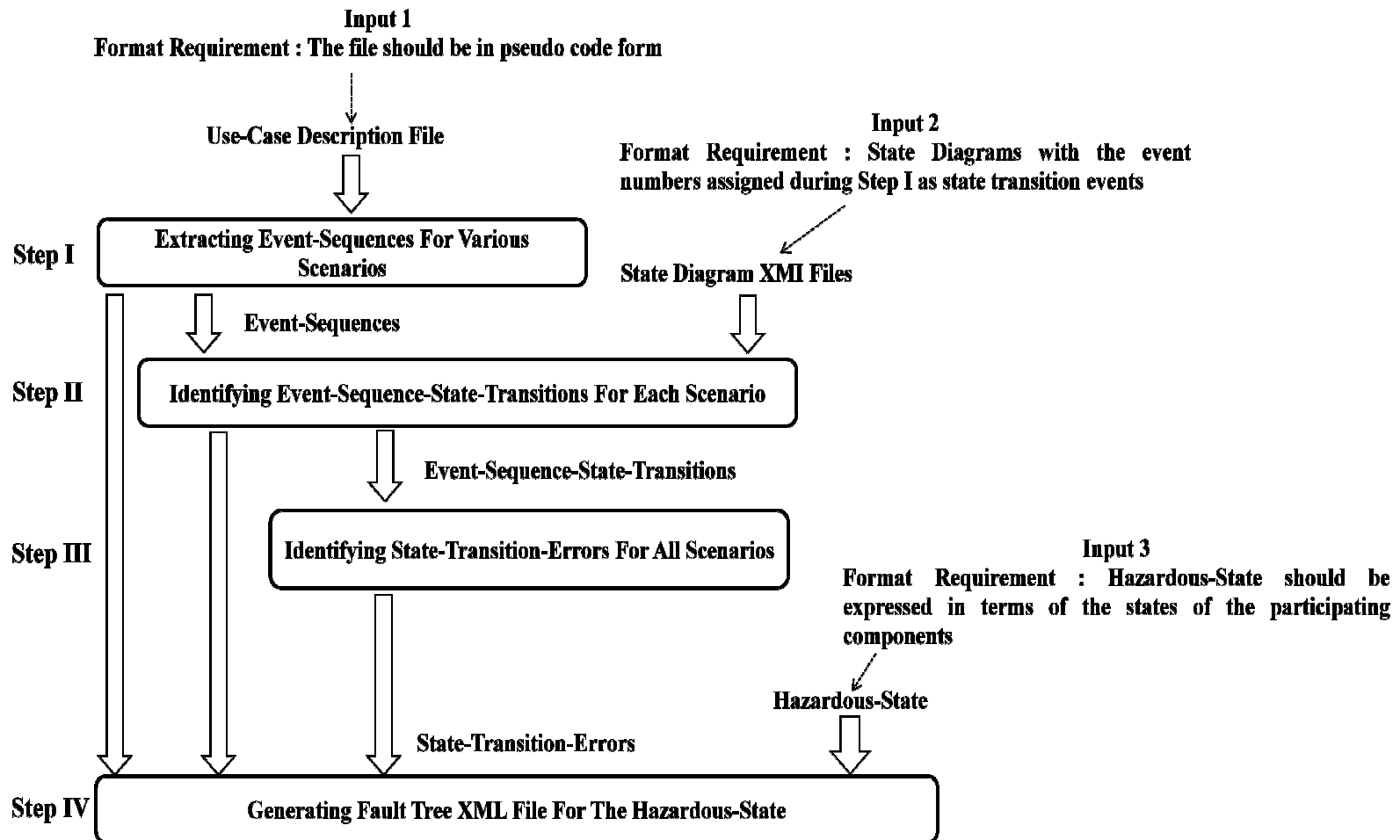


Figure 3.1: Overview of the Proposed SFTA Algorithm

3.5 SFTA ALGORITHM

In order to explain proposed SFTA algorithm in detail in the following sections, a simple use-case model as shown in Figure 3.2 is used. Assume an actor named 'A' and a use-case named 'doOperation'. The formal textual description of the 'doOperation' use-case consisting a loop condition is shown in Figure 3.3. This description is used throughout this section to illustrate the various steps of the proposed SFTA algorithm.

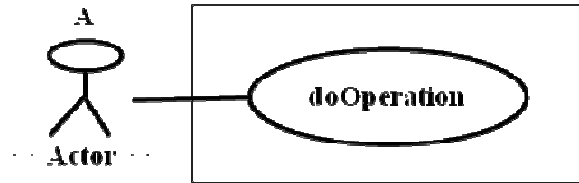


Figure 3.2: An Example Use-Case Model

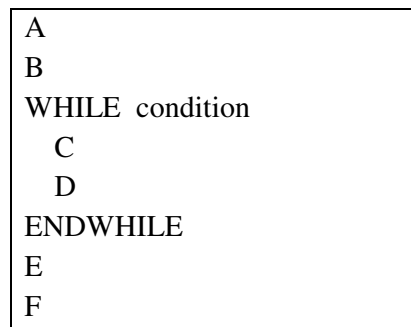


Figure 3.3: Use Case Description File For the 'doOperation' Use-Case of Figure 3.2

3.5.1 Step 1: Extracting Event-Sequences for Various Scenarios

In the first step all executable paths of a given use-case, known as scenarios, are identified. This creates an instance of a table named '*Event-Sequence*' corresponding to each scenario of the given use-case functionality.

The following tasks are carried out in sequence to complete the task.

- Each executable event mentioned in the use-case description file is assigned a unique identifier in the form of event number (*Event#*). It should be noted that not every line of use-case description file represents an executable event. For example, 'ENDWHILE' line in the use-case description file of Figure 3.3 is a non-executable event and hence is not assigned any event number.
- The precondition value is computed for each executable event. This value gives the information about the event sequence that is executed before the event.

- A logical time value is assigned to each executable event which represents its sequence of execution in the scenario event-sequence.

The scenario extraction process is carried out in two sub steps.

Step I (a) Creating a table with Event-Details

This sub-step creates a table named ‘*Event-Details*’ whose structure has three fields as (i) ‘Event#’, (ii) ‘Event-Name’ and (iii) ‘Event-Label’. In this step, each executable event is assigned a unique event number (‘*Event#*’). The ‘Event-Name’ is assigned for each event from the use-case description file.

The ‘Event-Label’ field is assigned as follows:

Suppose an event ‘ E_K ’ (‘*Event#*’) is assigned an ‘Event-Label’ value as ‘ E_L ’, where ‘ E_L ’ has a form: $E_1, E_2, \dots, E_{K-1}, E_K$ and it indicates that the event sequence (E_1, E_2, \dots, E_{K-1}) has been executed before E_K . The first event in any scenario is assigned an ‘*Event-Label*’ value equal to its own event number (‘*Event#*’). If there exists an executable event ‘ E_N ’ such that the execution control can reach ‘ E_N ’ via various possible paths, then the ‘Event-Label’ value of ‘ E_N ’ contains the event sequences of all such paths concatenated by an ‘OR’ operator. Apart from the executable events, the ‘Event-Label’ value is also computed for two non-executable events namely ‘ENDWHILE’ and ‘ENDIF’ also. It is to be noted that no ‘Event-Label’ value is computed for ‘ELSE’.

The process of assigning values to ‘Event-Label’ fields of various executable events is illustrated in Figure 3.4 for the use-case description as shown in Figure 3.3.

The executable events namely A, B, WHILE condition, C, D, E and F are assigned unique event numbers from E1 to E7. The ‘Event-Label’ value is computed for ‘ENDWHILE’ event also but no event number is assigned to ‘ENDWHILE’ because it is a non-executable event. Event labels are assigned as unique sequence numbers in the form E_1, E_2, \dots, E_K . The event E3 (WHILE condition) is a loop event and leads to two paths. The first path represents the case when the result of E3 is true (represented by E3(T)). The second path represents the case when the result of E3 is false (represented by E3(F)). The value assigned to ‘Event-Label’ field of event E6 contains two event sequences concatenated via an ‘OR’ operator as ‘ $\{E_1, E_2, E_3(F), E_6\}$ OR $\{E_1, E_2, E_3(T), E_4, E_5, E_5, E_6\}$ ’. It indicates that there exist two execution paths/routes via which a control can reach event E6. Similarly, the value assigned to ‘Event-Label’ field of event E7 also contains an ‘OR’ operator.

Step I(a) : Assign Event Numbers & Event Labels

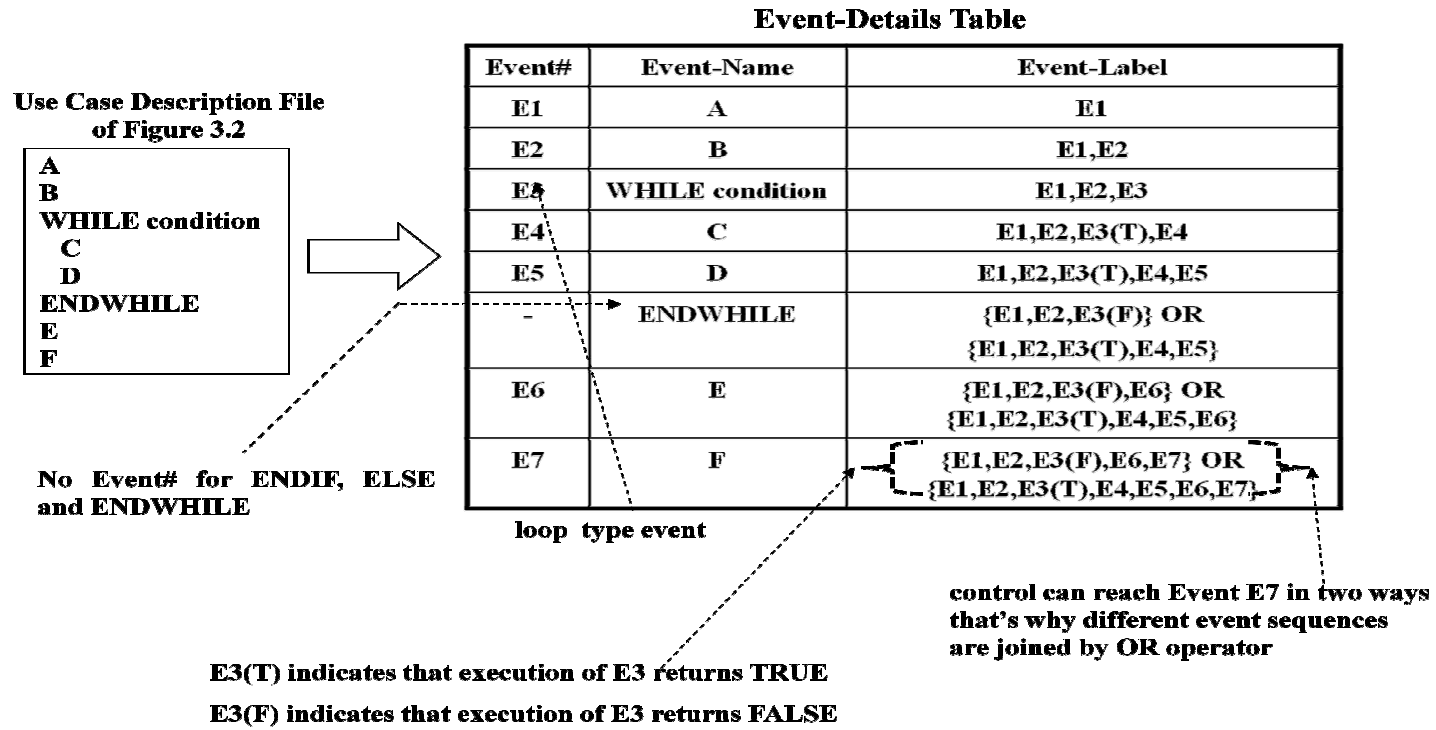


Figure 3.4: Operational Details of Step I(a)

The pseudo code for executing sub-step I(a) is given below in the form of a procedure named *assignLabels()*.

```

Procedure assignLabels()
Input      Use-Case Description File
Output    Event-Details
Variable(s) Used
String    ifLabel, startLabel, whilestartLabel, currentLabel=null,
            /* ifLabel → Label assigned to IF*/
            /*startLabel → label assigned to start of IF*/
            /*whilestartLabel → label assigned to start of while loop*/
            /*currentLabel → label assigned to current executable event*/
Stack     ifStack, whileStack
            /* Stack is a last-in first-out (LIFO) type data structure in which element added in last will be deleted
            first */
            /* ifStack → stack for IF block */
            /* whileStack → stack for while block */
Boolean   ElseFlag /* elseFlag is a Boolean type flag which represents whether IF has an associated ELSE or not
            */

/* Pseudo Code Description */
1. Create 'Event-Details' table with a structure as shown in Figure 3.2
2. FOR each executable event in the input file;
    (i) Assign Event# (Unique Event number for the event);
    (ii) Append the values of Event#, event-name in the Event-Details table;
ENDFOR
3. FOR each row of the Event-Details table created above
    Case: row contains an 'IF'
        set elseFlag = false;
        set startLabel = currentLabel;
        set currentLabel = currentLabel+event+'(T)';
        push startLabel onto ifStack; /* push is the name of add operation for Stack */
    Case: row contains an 'ELSE'
        pop top element from ifStack; /* pop is the name of delete operation for Stack */
        set elseFlag = true; /* IF has an associated ELSE option */
        set ifLabel = currentLabel;
        set currentLabel = startLabel;
        set currentLabel = currentLabel+'(F)'; /* '(F)' represents false condition */
    Case: row contains an 'ENDIF'
        pop top element from stack;
        IF elseFlag = true THEN
            set currentLabel = ifLabel +currentLabel;
        ELSE
            set currentLabel = currentLabel+startLabel+'(F)';
        ENDIF
        set elseFlag = false;
    Case: 'event' is WHILE type
        set whilestartLabel = currentLabel;
        set currentLabel = currentLabel+event+'(T)';
        push whileStartLabel on to whileStack;
    Case: 'event' is ENDWHILE type
        pop top element from whileStack;
        set currentLabel = currentLabel + whilestartLabel + event + "(F)";
    Default:
        IF currentLabel = null THEN
            currentLabel = event;
        ELSE
            currentLabel = currentLabel + event;
        ENDIF
ENDFOR

```

Step I (b) Extracting the Event-Sequence for each Scenario

The event sequence of a particular scenario is extracted from the event label values assigned to various executable events in the previous sub-step. An event label value ' E_L ' assigned to an event ' E_K ' represents a potential scenario event sequence, if no other event's event label value contains E_L . If E_L contains an 'OR' operator then each event sequence concatenated via an 'OR' operator represents the scenario event sequence.

The details of this step are illustrated in Figure 3.5 for the example use-case. The 'Event-Details' table as shown in Figure 3.4 is the input in this sub-step. The 'Event-Label' value of event E7 is '{E1, E2, E3(F), E6, E7} OR {E1, E2, E3(T), E4, E5, E6,E7}' and this value is not contained in the 'Event-Label' value of any other event. So, the 'Event-Label' value of event E7 represents a scenario event sequence. As this value contains only one 'OR' operator, it represents two scenario event sequences and these are event sequences are {E1, E2, E3, E4, E5, E6, E7} and {E1, E2, E3, E6, E7}. The values of the 'Logical_Time' fields of the events in an event sequence, are assigned sequentially as 1,2,3..and so on.

Similarly, for another use-case as shown in Figure 3.6, three scenarios with event sequences as {E1, E2, E3, E4, E5, E6, E8}, {E1, E2, E3, E4, E7, E8} and {E1, E2, E3, E8} are identified and logical time values are assigned as per above mentioned criteria.

The pseudo code description of this sub-step is given below in the form of a procedure named *createEventSequenceTables()* as follows:

Procedure *createEventSequenceTables()*
Input *Event-Details table of Step I(a)*
Output *Event-Sequence table(s) for each scenario*
/* Pseudo Code Description */

1. *Identify the number of possible scenarios from the Event-Details table of Step I(a)*
2. *Create and Populate Event-Sequence table for each such scenario*

Step I(b) : Scenario Extraction Process

Event-Details Table as Input

Event#	Event-Name	Event-Label
E1	A	E1
E2	B	E1,E2
E3	WHILE condition	E1,E2,E3
E4	C	E1,E2,E3(T),E4
E5	D	E1,E2,E3(T),E4,E5
-	ENDWHILE	{E1,E2,E3(F)} OR {E1,E2,E3(T),E4,E5}
E6	E	{E1,E2,E3(F),E6} OR {E1,E2,E3(T),E4,E5,E6}
E7	F	{E1,E2,E3(F),E6,E7} OR {E1,E2,E3(T),E4,E5,E6,E7}



Event-Sequence Table For Scenario I

Event #	Precondition	Event-Name	Logical_Time
E1	-	A	1
E2	E1	B	2
E3	E1,E2	WHILE condition	3
E4	E1,E2,E3(T)	C	4
E5	E1,E2,E3(T),E4	D	5
E6	E1,E2,E3(T),E4,E5	E	6
E7	E1,E2,E3(T),E4,E5,E6	F	7

Event-Sequence Table For Scenario II

Event #	Precondition	Event-Name	Logical_Time
E1	-	A	1
E2	E1	B	2
E3	E1,E2	WHILE condition	3
E6	E1,E2,E3(F)	E	4
E7	E1,E2,E3(F),E6	F	5

Figure 3.5: Scenario Extraction (Step I(b)) for Use-Case Description of Figure 3.3

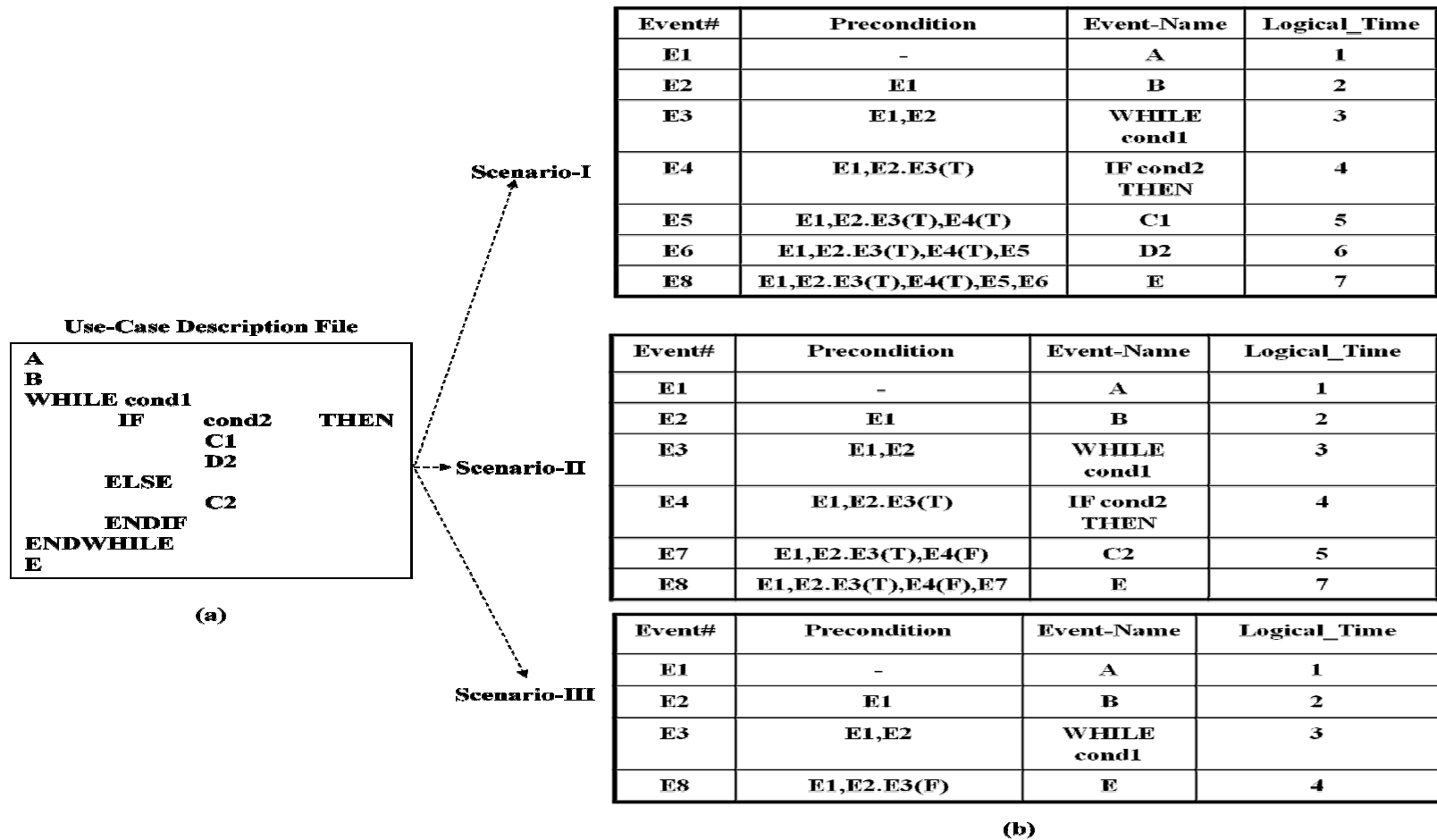


Figure 3.6: Scenario Extraction Process Illustration for Use-Case Description Shown in (a)

3.5.2 Step II: Identifying Event-Sequence-State-Transitions for each Scenario

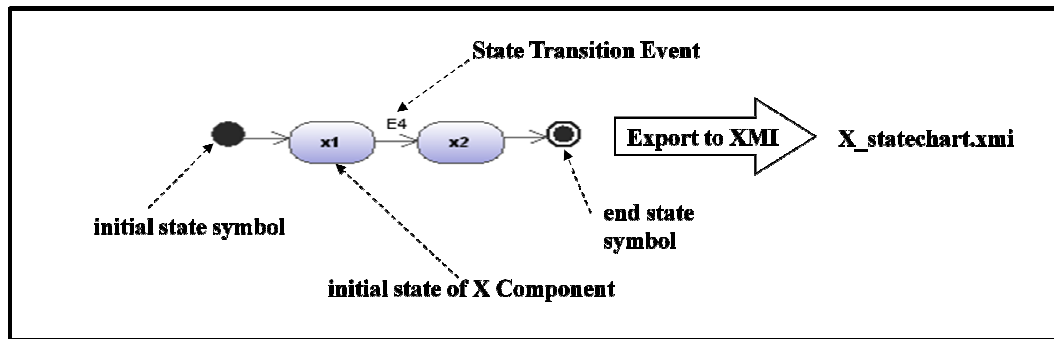
Using various ‘*Event-Sequence*’ tables created in Step I and the state diagram XMI files of the participating components as inputs, events of various scenarios are mapped against the states of the participating components to identify ‘*Event-Sequence-State-Transitions*’ for each scenario. The following two conditions should be satisfied before the start of this step.

- (i) The state diagrams are drawn by using the event numbers (assigned to various executable events in Step I) as state transition events.
- (ii) If the state transition pattern of a component is uniform across all the scenarios, then only one state diagram is drawn for that component. But, if a state transition pattern for a component is different in different scenarios, then a separate state diagram for that component is drawn for each scenario.

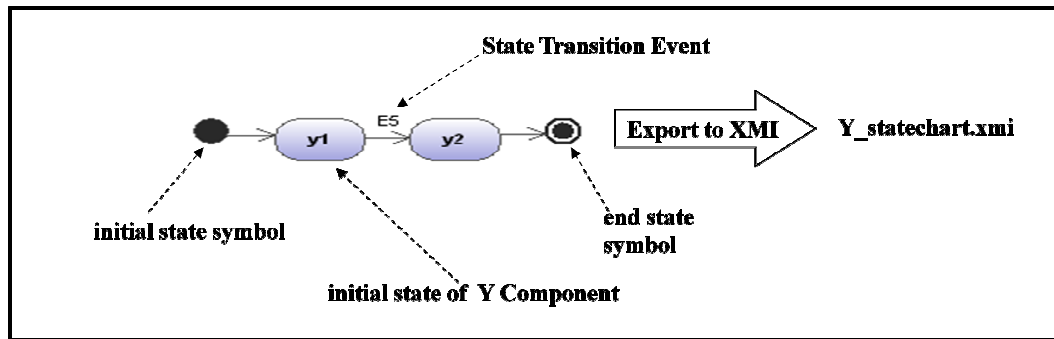
This step creates an instance of a table named ‘*Event-Sequence-State-Transition*’ table corresponding to each instance of ‘*Event-Sequence*’ table created and populated in Step I. The structure of the ‘*Event-Sequence-State-Transition*’ table depends upon the number of components (not the number of state diagrams) for which the state diagrams are supplied as inputs in this step. The first field of this table is *Event#*(Event Number) and every other field represents the name of the component for which a state diagram is drawn.

Suppose two components, namely ‘X’ and ‘Y’ are participating in the use-case functionality of Figure 3.3. Recall that the same use-case functionality is used as an input in Step I. Assume that the valid states of components ‘X’ and ‘Y’ are {x1 and x2} and {y1 and y2}, respectively. The UML state diagrams drawn for both these components are shown in Figure 3.7. The initial state of ‘X’ component is ‘x1’ whereas the initial state of ‘Y’ component is ‘y1’. The execution of event E4 (event C as shown in Figure 3.4) changes the state of ‘X’ component to ‘x2’. Similarly, the execution of event E6 (event E of Figure 3.4) changes the state of ‘Y’ component to ‘y2’.

Taking the ‘*Event-Sequence*’ tables as shown in Figure 3.5 and the UML state diagrams of components ‘X’ and ‘Y’ as shown in Figure 3.7 as inputs, ‘*Event-Sequence-State-Transitions*’ are identified for two scenarios as shown in Figure 3.8. These are saved in ‘*Event-Sequence-State-Transition*’ table in Figure 3.8(b) with the structure of three fields as (i) *Event#*, (ii) X and (iii) Y because the state diagrams are drawn for two components namely ‘X’ and ‘Y’.



(a) State Diagram for X Component



(b) State Diagram for Y Component

Figure 3.7: State Diagrams for two Components X and Y

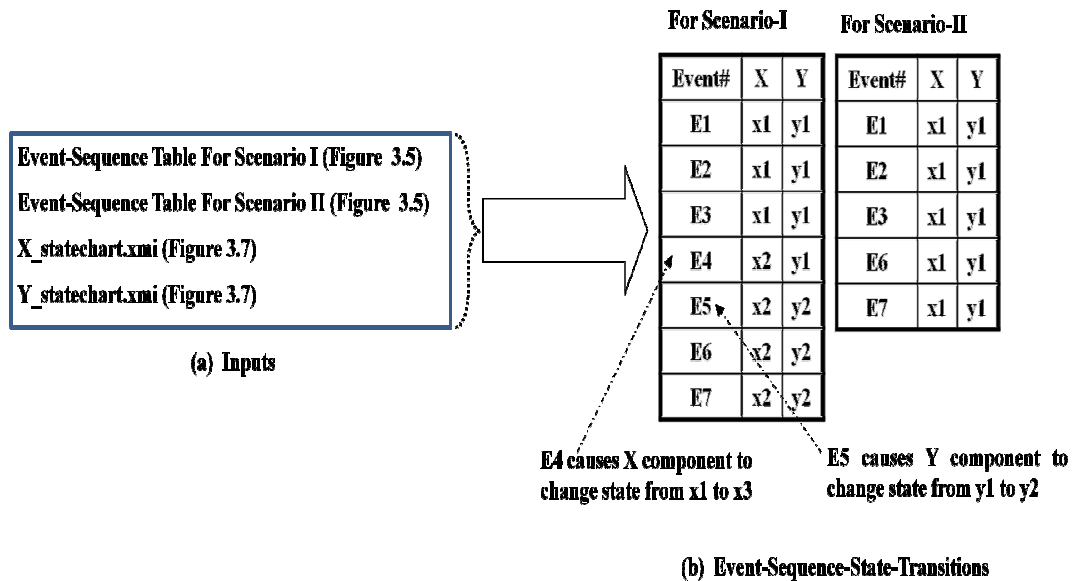


Figure 3.8: Event-Sequence-State-Transition Tables for Two Scenarios

It is to be noted that corresponding to event E4 of ‘*Event-Sequence-State-Transition*’ table for Scenario I, as shown in Figure 3.8(b), the execution of event E4 causes the state of X changed to x2 from x1 and similarly the execution of event E5 causes the state of Y changed to y2 from y1.

The pseudo code for identifying all ‘*Event-Sequence-State-Transitions*’, in the form of a procedure named ‘*createStateTransitionTables()*’ is given below.

```

Procedure    createStateTransitionTables()
Input       State Chart XMI file(s) and Event-Sequence tables of Step I
Output      Event-Sequence-State-Transition table corresponding to each Event-Sequence
            table
/* Pseudo Code Description*/
1.FOR each Event-Sequence table created in Step I of the approach
    Create a corresponding instance of Event- Sequence-State-Transition table;
    ENDFOR
2.FOR each Event-Sequence-State-Transition table created
    FOR each component-name column of the Event-Sequence-State-Transition table
        select the associated state diagram XMI file for the component;
        read the initial_state for the component from XMI file;
        FOR each event number ‘E’ of the selected Event-Sequence-State-Transition table
            scan the selected XMI file for state transition corresponding to ‘E’;
            IF ‘E’ is responsible for any state transition for the component THEN
                read next_state_transition for component-name from the XMI file;
                set the new value of initial_state as next_state_transition;
                update component-name column with next_state_transition;
            ELSE
                update component-_name column with initial_state;
            ENDIF
        ENDFOR
    ENDFOR
ENDFOR

```

3.5.3 Step III: Identifying State-Transition-Errors for all scenarios

The objective of this step is to identify state-transition-error and each one is saved in a single instance of a table named ‘*State-Transition-Error*’. This step uses the ‘*Event-Sequence-State-Transition*’ tables instantiated in Step II as inputs and records those errors which, if allowed to occur can prevent the component from making its desired state transition leading to a faulty operation. In order to carry out this task, every component column of every ‘*Event-Sequence-State-Transition*’ table is scanned to locate various events where the selected component is changing its state. For example, ‘*Event-Sequence-State-Transition*’ table of scenario I, as shown in Figure 3.8(b), indicates that

the component 'X' is changing its state from 'x1' to 'x2' during the execution of event 'E4'. The errors that can occur during the execution of event 'E4' and can prevent the component 'X' from making its desired state change (i.e. Component 'X' remains in the state 'x1' and does not successfully change its state to 'x2') are identified as state-transition-errors and are recorded in a 'State-Transition-Error' table. Each such state related error is identified by a unique error number (Error#), error name (Error-name), event number where it occurs (Event#) and its final effect (Effect). There are two types of state related errors that can prevent a component from making its required state change. Recall that each state transition event is an executable event. The first type of error belongs to the '*software-control*' category where the state transition event fails to execute at all. The second type of error represents the situation where a fault occurs in the component itself. For example, consider the execution of event 'E4' that causes a change in the state of 'X' component from initial state 'x1' to state 'x2' (Figure 3.8(b)). The first type of error is the situation where the event 'E4' fails to execute at all and the second type of error is the situation where the error occurred in the component 'X' itself.

The pseudo code for identifying various state transition errors is given below.

```

Procedure    identifyStateTranistionErrors()
Input(s)    Event-Sequence-State-Transitions of Various Scenarios
Output      State-Transition-Errors
FOR each Event-Sequence-State-Transitions
FOR each event in the Event-Sequence-State-Transitions
    IF event is changing the state of a component THEN
        record two state related errors corresponding to the event and assign a
        uniuue error number to each error;
    ENDIF
ENDFOR
ENDFOR

```

For the 'Event-Sequence-State-Transitions' as shown in Figure 3.8, the execution of this step results in the instantiation of 'State-Transitions-Errors' as shown in Figure 3.9(b). The 'State-Transitions-Errors' table, as shown in Figure 3.9, has two state transition events ('E4' and 'E5') and four states related errors with error numbers as ER1, ER2, ER3 & ER4. The number of 'State-Transitions-Errors' is twice the number of state transition events since two state-related errors are identified corresponding to each state transition event.

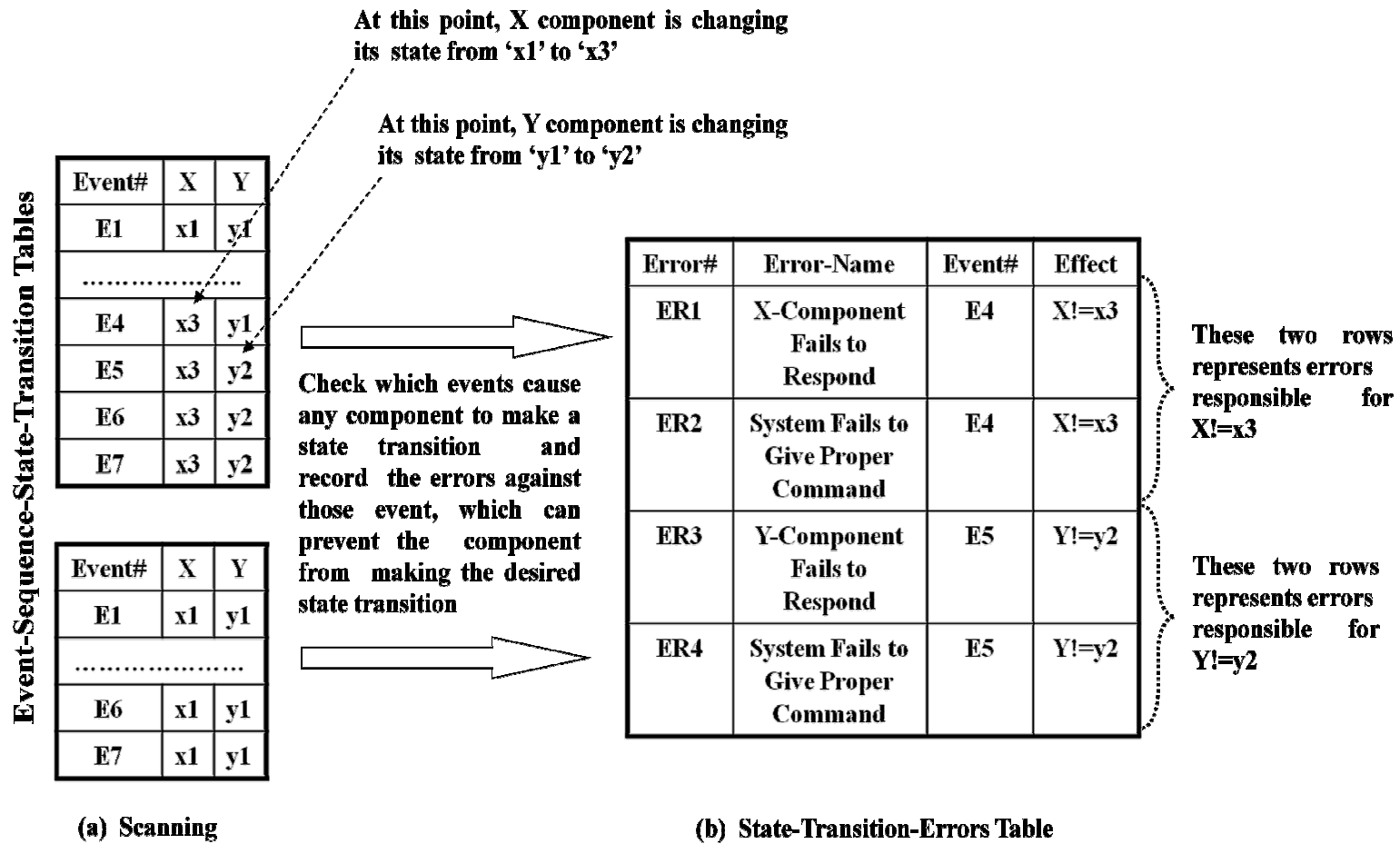


Figure 3.9: Populating State-Transition-Error Table Generated From Tables of Figure 3.8

3.5.4 Step IV: Generating Fault Tree XML File

The software fault tree for a given hazardous-state of the system is constructed in this step. The output of this step is in the form of one or more fault tree XML files. The actual fault trees can be constructed from these XML file(s) by using an available fault tree creation tool named FaultCAT (FaultCAT, 2003) in the next step. The outputs of Steps I, II and III and the hazardous-state are required inputs to this step.

The first task is to define the syntax of the hazardous-state. Consider some components $\{X_1, X_2, \dots, X_n\}$ and their respective states $\{x_1, x_2, \dots, x_n\}$. The hazardous-state of the system is expressed either in atomic form or in composite form as defined in Section 3.3. Assume the components $\{X_1, X_2, \dots, X_n\}$ make the state transitions as $\{x_1, x_2, \dots, x_n\}$ at logical time values $\{t_1, t_2, \dots, t_n\}$ respectively.

The composite hazardous-state can be categorized in two types as explained below.

(i) Type I composite hazardous-state

If a component is not able to change its state, then it is represented by a negation state symbol (\neq) and alternatively if a component is able to change its state successfully, then it is represented by a true state symbol ($=$). The state of the first component (i.e. X_1) of this hazard type should have a negation type symbol (\neq) whereas the states of other components can use either a negation (\neq) or a true ($=$) type symbol. This type of hazard will be considered as valid if the condition ' $t_1 < t_2 \dots < t_n$ ' holds true. This type of hazardous-state indicates the situation where the first component X_1 (to state ' x_1 ') fails to change its expected state whereas the other components ($X_2 \dots, X_n$) either failed or succeeded in making their respective state transitions.

The generalized form for Type I composite hazardous-state is as follows:

$$\text{Type 1: } X_1 \neq x_1 \text{ AND } X_2 [! = \text{ or } =] x_2 \dots \text{ AND } X_n [! = \text{ or } =] x_n$$

(ii) Type II composite hazardous-state

This type of hazard is used to represent the situation where it is considered dangerous from the system perspective, to have the components $X_1, X_2 \dots X_n$ in states $x_1, x_2 \dots x_n$ respectively.

For example, in an elevator control application, at any point of time, it is dangerous to have the component 'door' in 'opened' and the component 'motor' in 'moving' states ($door=opened \text{ AND } motor=moving \text{ OR alternatively } motor=moving \text{ AND } door=opened$). Similarly, in a rail track door controller application, it is dangerous to have the component 'track_door' in 'opened' and the component 'track_signal' in 'green' states respectively ($track_door=opened \text{ AND } track_signal = green$).

A system encounters Type II hazardous situations only when some of the components involved in the hazardous-state fail to make their required state transitions and remain in their previous state. For ECS example, consider at time ' t_1 ' the state(s) of the components 'door' and 'motor' are 'opened' and 'stopped' respectively. If at time ' t_2 ', the component 'motor' is changing its state to 'moving' and ' t_1 ' is less than ' t_2 ', then, there exists an event ' E_x ' at time ' t_3 ' such that $t_1 < t_3 < t_2$ where the component 'door' is supposed to make a change in its state from 'opened' to 'closed'. If this expected state change in 'door' component fails to occur, then the 'door' component will remain in the 'opened' state. Hence, the system will encounter a hazardous-state ' $door=opened \text{ AND } motor = moving$ ' at time ' t_2 ' when the state transition for the 'motor' component to 'moving' state will occur successfully.

The elevator control system can encounter the same hazardous-state via an alternative scenario also. Consider an alternative scenario, where at time t_1 the states of the components 'motor' and 'door' are 'moving' and 'closed' respectively. If at time ' t_2 ', the component 'door' is changing its state to 'opened' and t_1 is less than t_2 , then, there exists an event E_x at time t_3 such that $t_1 < t_3 < t_2$ where the 'motor' component is supposed to make a change in its state from 'moving' to 'stopped'. If this expected state change for the 'motor' component fails to occur, then the 'motor' component will remain in the 'moving' state. Hence, the system will encounter hazardous-state ' $door=opened \text{ AND } motor = moving$ ' at time t_2 when the state transition for the 'door' component to 'opened' state will occur successfully. From the above discussion, it can be concluded that a type 2 hazardous-state $door=opened \text{ AND } motor = moving$ can be interpreted as type 1 hazardous-state either as ' $door! =closed \text{ AND } motor = moving$ ' [provided $logical_time(door=closed) < logical_time(motor=moving)$] or as ' $motor!=stopped \text{ AND } door = opened$ ' [provided $logical_time(motor=stopped) < logical_time(door=opened)$].

The generalized form for Type II composite hazardous-state is as follows:

$$\text{Type 2: } X_1 = x_1 \text{ AND } X_2 = x_2 \dots \text{AND } X_n = x_n$$

The validity condition for this type of hazardous-state is: $t_1 \neq t_2 \dots \neq t_n$.

The fault tree for an atomic hazardous-state can be drawn by taking the ‘*State-Transition-Errors*’ of Step III as an input. But the fault trees for composite hazardous-states are constructed by using the outputs of Step(s) I, II and III along with the hazardous-state as inputs.

The fault tree constructed for the hazardous-state ‘ $X \neq x_2 \text{ AND } Y = y_2$ ’ is illustrated in Figure 3.10. The ‘*Event-Sequences*’ (Figure 3.5), the ‘*Event-Sequence-State-Transitions*’ (Figure 3.8) and ‘*State-Transition-Errors*’ (Figure 3.9) are used as the inputs in the construction of this fault tree.

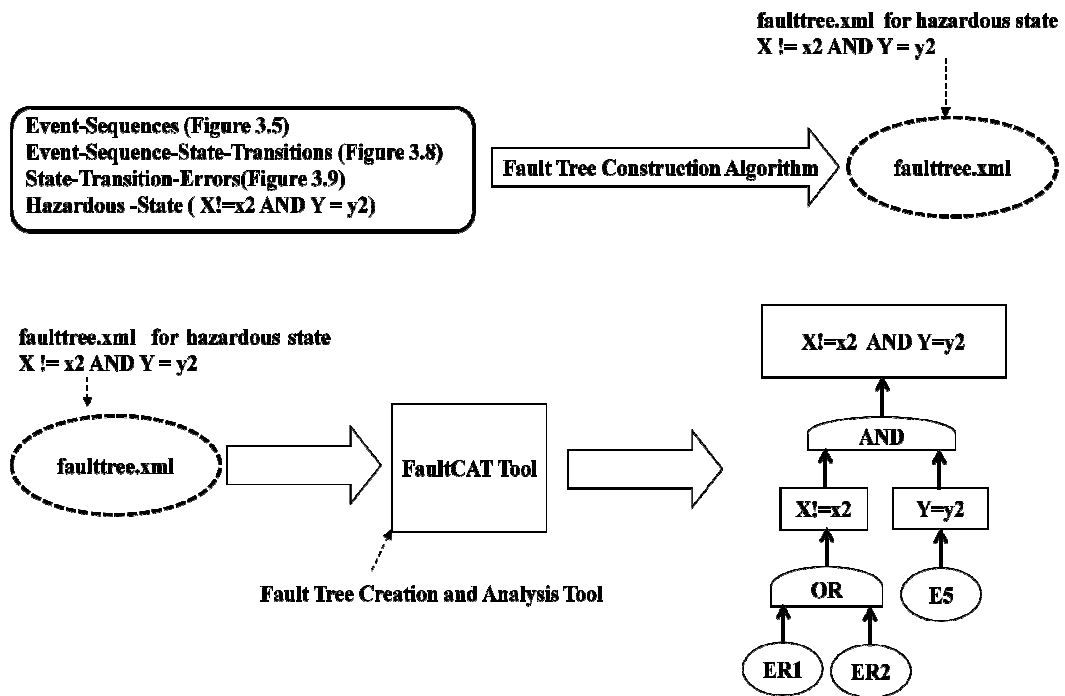


Figure 3.10: Illustration of Software Fault Tree Construction Process

The algorithm for constructing the fault tree for Type 1 hazardous-state has the following four steps.

- (i) Selecting of scenarios where the given hazardous-state can occur and parsing of the erroneous states of the participating components such as $X_1 \neq x_1$, $X_2 = x_2$, $X_3 = x_3$, etc.
- (ii) Constructing the fault tree for the first erroneous state $X_1 \neq x_1$.
- (iii) Constructing a fault tree for each of the successive erroneous states recursively
- (iv) Combining all the constructed fault trees via AND gates as required.

If the selected composite hazardous-state can occur in more than one scenario, then the same steps are applied recursively for each scenario. The output *faulttree.xml* file is generated separately for each scenario.

The pseudo code of the fault tree construction process for Type I hazardous-state in the form of a procedure named '*createFaultTree_TypeI()*' is given below.

Procedure *createFaultTree_TypeI()*

Input(s) *Event-Sequence Tables (Step I output), Event-Sequence-State-Transition Tables (Step II output), State-Transition-Error Table (Step III output) and hazardous-state*

Output *Fault Tree XML (faulttree.xml) files*

Variable(s) used in the pseudocode description

root_node *The hazardous-state (for which the software fault tree is to be constructed)*

state-errorList *The list of erroneous states for various components mentioned in the hazardous-state, e.g. if the hazardous-state is $X_1 \neq x_1$ AND $X_2 \neq x_2$ AND $X_3 = x_3$ then the state-errorList will contain the values as $\{X_1 \neq x_1, X_2 \neq x_2, X_3 = x_3\}$*

hazard_scenarioList *This represents the list of all 'Event-Sequence-State-Transition' tables where the selected hazardous-state can occur*

previous_change_event *The event involved in state change of the previous erroneous state*

current_change_event *The event involved in state change of the current selected erroneous state*

/* Pseudocode Description */

1. Initialize the data structures

1.1 Set root_node = hazardous-state

1.2 Parse and extract various erroneous states from the given hazardous-state and initialize state-errorList

1.3 Select the 'Event-Sequence-State-Transition' tables where the hazardous-state can occur & initialize hazard_scenarioList

2. FOR each 'Event-Sequence-State-Transition' table in hazard_scenarioList (constructed in step 1.3)

2.1 Create an associated faulttree.xml file

/* Create Fault Tree for the First Erroneous State $X_1 \neq x_1$ */

2.2 Create Fault Tree for the erroneous state $X_1 \neq x_1$ using errors from State-Transition-Error table and write it in the xml file and set previous_change_event = {event where an effect $X_1 \neq x_1$ has occurred}

/* Create Fault Tree for each of the remaining Erroneous States */

2.3 FOR each remaining erroneous state in the 'state-errorList'

Case: if erroneous state is Negation Type (! =)

Create Fault Tree for the erroneous state using errors from State-Transition-Error table and write it in the xml file and set previous_change_event = {event where the current erroneous state has occurred}

Case: if erroneous state is True Type (=)

a. Set the value of current_change_event = {event where the current erroneous state has occurred}

b. Select the event sequence that is executed after the previous_change_event to the current_change_event

c. Create a basic error event for each event selected in the previous step and write it in the xml file

ENDFOR /*End of Step 2.3*/

/* Join the Created Fault Trees via an AND gate */

2.4 Join the trees created at step(s) 2.2 and 2.3 as via an AND gate with root_node as the output of this AND gate

ENDFOR /*End of Step 2*/

To construct a fault tree for Type 2 hazardous-state, firstly, the given Type 2 hazardous-state is converted into Type 1 hazardous-state and then the Type 1 fault tree construction procedure is applied on the converted hazardous-state.

The procedure for drawing the fault tree for Type 2 hazardous-state has followed four steps.

- (i) Selecting the scenarios where the given hazardous-state can occur.
- (ii) Converting Type II hazardous state into Type I hazardous state, according to the selected scenario as follows
 - (a) Sort the component states (involved in the hazardous-state) on the logical time value of their occurrence
 - (b) Identify the state (say 'x') which the first component (as per sorted list) fails to experience
 - (c) Construct the converted hazardous-state by using '!= ' symbol for the state of the first component and '=' symbol for the states of other components.
- (iii) Using Type I procedure to constructing the fault tree for the converted hazardous state
- (iv) If number of scenarios selected in step (i) above are more than one, then combining all the constructed fault trees via OR gates, as required.

The pseudo code of the fault tree construction process for Type II hazardous-state in the form of a procedure named '*createFaultTree_Type2()*' is given below.

Procedure createFaultTree_Type2()

Input(s) Event-Sequence Tables (Step I output), Event-Sequence-State-Transition Tables (Step II output),

State-Transition-Error Table (Step III output) and hazardous-state

Output Fault Tree XML (faulttree.xml) files

Additional variable(s) used in the pseudo code description of this type

<i>component_event_List</i>	<i>Every element of this list is of the form {a,b,c,d} where 'a' is the name of the component, 'b' represents the selected state of the component 'a', 'c' represents the event when the component 'a' is in the selected state 'b' and 'd' represents the logical time of event 'c'</i>
<i>component_change_last</i>	<i>Component whose state is changed in the last (as per logical time value)</i>
<i>state_change_last</i>	<i>State of the component_change_last i.e. the state changed by the last component</i>
<i>event_change_last</i>	<i>Event responsible for the state of the component_change_last</i>
<i>time_change_last</i>	<i>Logical time when event_change_last has occurred</i>
<i>converted_hazardous_state</i>	<i>Represents the transformed hazardous state and its current value is null</i>

/* Pseudocode Description */

- 1. Initialize the data structures /*Same as for Type I pseudocode description*/*
- /* Convert the given Type 2 hazardous-state into Type 1 hazardous-state */*

```

2. FOR each 'Event-Sequence-State-Transition' table in hazard_scenarioList [populated in Step 1]
  2.1. FOR I = 1 to 'n' /* where 'n' is the number of erroneous states in the given hazardous-state */
    Search for entry  $X_{[I]} = x_{[I]}$  and add the component  $X_{[I]}$ , the state  $x_{[I]}$ , an event 'E' where  $X_{[I]} = x_{[I]}$ 
    has occurred and logical_time of E into component_event_List;
  ENDFOR
  2.2. Sort the component_event_List on logical time value and initialize the values for variables
  component_change_last, event_change_last, state_change_last and time_change_last
  2.3. For each element in component_event_List except the last component i.e. component_change_last
    2.3.1. Find the state transition (say a) for the current component (say X) that has occurred
    between the current event (given by the current element of component_event_List) and
    the event_change_last
    2.3.2. Negate the state as  $X \neq a$  /* Component X fails to make its desired state change and
    remains in previous state*/
    2.3.3. IF converted_hazardous_state is null THEN
      Set converted_hazardous_state =  $(X \neq a)$ ;
    ELSE
      Set converted_hazardous_state = converted_hazardous_state + AND +  $(X \neq a)$ ;
      /* '+' is string concatenation operator*/
    ENDIF
  ENDFOR /* End of Step 2.3*/
  2.4. Set converted_hazardous_state = converted_hazardous_state + AND +
  component_change_last+ state_change_last
  /* Invoke Type1 Procedure for theconverted_hazardous_state */
  2.5. Use Type 1 Procedure to construct the fault tree for the converted_hazardous_state
ENDFOR /*End of Step 2*/
3. IF the number of 'Event-Sequence-State-Transition' tables selected in Step 2 is more than one
  THEN
    Merge the software fault trees created in Step 2 via an OR gate and set the hazardous-state as the
    root node of the merged tree;
  ELSE
    Create a wire gate with fault tree of step 2 as input and the hazardous-state  $X_1 = x_1$  AND  $X_2 = x_2 ..$ 
    AND  $X_n = x_n$  as an output;
  ENDIF /*End of Step 3*/

```

3.5.5 Step V: Drawing Fault Tree From XML File

This step constructs the fault tree by giving the XML file (created in Step IV above) as an input to a fault tree creation tool named FaultCAT (FaultCAT, 2003) as shown in Figure 3.10.

3.5.6 Salient Features and Time Complexity of the SFTA Algorithm

The software fault tree construction process is easily scalable to any number of state variables for both types of hazardous-states. The algorithmic time complexity i.e. the running time of the first four steps of the SFTA algorithm is given in the following sections. Note that the fifth step simply constructs the fault tree in graphical form and no computation is done in this step.

(a) Time Complexity of Step I

The running time, i.e. the algorithmic time complexity of the Step I (a) is of the order of 'O(N₁)' where 'N₁' is the number of executable events in the given use-case description file.

The algorithmic complexity of the Step I(b), is approximately of the order of 'O(N₂)' where 'N₂' is the number of event sequence tables created.

Hence, the total execution time for Step I is 'O(N₁) + O(N₂)'.

(b) Time Complexity of Step II

The running time (i.e. Algorithmic time complexity) of Step II is 'O(N₃ × N₄ × N₅)' where 'N₃' is the number of 'Event-Sequence-State-Transitions', 'N₄' is the number of components for which a state diagrams are drawn and 'N₅' is the average number of executable events in each 'Event-Sequence'.

(c) Time Complexity of Step III

The algorithmic time complexity of Step III is of the order of 'O(N₄ × N₅)' where the meaning of 'N₄' and 'N₅' are already explained in part 3.5.5 (b) above.

(d) Time Complexity of Step IV

The algorithmic time complexity of Step IV is of the order of 'O(N₆ × N₇)' where 'N₆' is the number of scenarios where selected hazardous-state can occur and 'N₇' is the number of erroneous states present in the selected hazardous-state.

The overall algorithmic time complexity of all the four steps of the SFTA algorithm is as follows:

$$[O(N_1) + O(N_2)] + [O(N_3 \times N_4 \times N_5)] + [O(N_4 \times N_5)] + [O(N_6 \times N_7)]$$

3.5.7 Formatting of Inputs

The proposed SFTA algorithm as described in Section 3.5 assumes that the three inputs namely (i) use-case description file, (ii) UML state diagrams of the participating components and (iii) the hazardous-state of the system, are supplied in some specific representations and these representation for each input is explained in the following sub-sections.

(a) Use-Case Description File Representation

The whole description of the selected use-case functionality is to be supplied as a single text file. The text within this file is to be expressed in the pseudo code form using structured English as follows:

- Every line mentioned under *event_details* part should represent a single event and this event can belong to any one of three categories (i) *normal event*, (ii) *conditional event* (IF-THEN-ENDIF, IF-THEN-ELSE-ENDIF) or (iii) a *loop event* (WHILE - ENDWHILE, DO WHILE - ENDDO). It should be noted that the description of each alternative flow is to be expressed using structured English constructs such as IF-THEN-ELSE-ENDIF etc.
- The basic-details part is inserted at the beginning of the file as comments using /* <text> */. Comments can be inserted/ added anywhere in the file to improve the clarity of the written text.
- An IF condition can exist without an ELSE option also but an ENDIF is mandatory for each IF block. Similarly, ENDWHILE and ENDDO are mandatory for each WHILE and DO WHILE blocks respectively.
- The words ELSE, ENDIF and ENDWHILE should appear on separate lines and should not be mixed with other events. These words are not considered as executable events.

There is no restriction on the size (*i.e. the number of lines*) of the use-case description file.

(b) State Diagram Representation

The proposed SFTA approach operates with the assumption that the state diagrams of the participating components are supplied in machine readable format *i.e.* XMI (XML Metadata Interchange). The Altova UML (Altova-UModel, 2014) tool has been used to draw the required state diagrams and each state diagram is exported to XMI (XML Metadata Interchange) format using the same tool. The main requirement is that *the UML state diagrams drawn for the participating components should use the unique event numbers assigned to various executable events as state transition events.*

(c) Hazardous-State Representation

The hazardous-state, for which a fault tree is to be constructed, is to be expressed in terms of the states of the participating components either in *atomic form* or in *composite form* as discussed in Section 3.3.

The next section demonstrates the step-by-step application of the algorithm on the use-case functionalities of two safety-critical applications.

3.6 MOTIVATING EXAMPLE 1:REQUEST ELEVATOR USE-CASE OF AN ELEVATOR CONTROL SYSTEM (ECS) APPLICATION

The use-case selected from an Elevator Controller System (ECS) application is '*Request Elevator*'. This use-case gets realized when any user from any floor presses the button to request an elevator to visit the requested floor number. Each floor button is assigned a unique number so that the pressing of the floor button also gives the information about the floor number of the building from where the button is pressed. The system has a device named 'Arrival Sensor' installed on each floor of the building. The role of this 'Arrival Sensor' component is to interrupt the system whenever an elevator is about to reach the respective floor number.

The formal textual description of the '*Request Elevator*' use-case is shown in Figure 3.11.

Step I: Extracting 'Event-Sequence' for Each Scenario

The application of the Step I(a) results in the extraction of 'Event-Details' as shown in Table 3.5. The 'Event-Label' value of ENDWHILE (last row of Table 3.5) is not a part of any other event label value and hence represents a potential Event-Sequence. There are seven event sequences joined by six 'OR' words in the 'Event-Label' value of this 'ENDWHILE'. So the application of Step I(b) results in seven 'Event-Sequences' and these event-sequences for Scenario 1 to Scenario 7 are shown in Table 3.6 to Table 3.12 respectively. [Note: The Event-Name column is not shown in the Event-Sequence tables to avoid replication of data].

```

/*****Basic Details*****/
/*      Use Case      :  Request Elevator      */
/*      Initiating Actor :  Elevator User      */
/*      Precondition   :                      */
/*****/
/*      Event Details                                */
user press elevator button
elevator button sensor reads the destination floor request and notifies it to system
system update the request
WHILE request queue is not empty
IF elevator is idle THEN
    Determine direction
    system commands the door to close
    IF door = closed THEN
        System commands to start the motor to move to the determined direction
    ENDIF
ELSE
    As the elevator is approaching the floors, floor sensor detects the floor # and notifies the
    system
    IF elevator has to stop at that floor THEN
        System commands the motor to stop
    IF motor= stopped THEN
        System commands the elevator door to open
    IF door=opened THEN
        system updates the request queue
    ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
ENDWHILE

```

Figure 3.11:Use Case Description File for ‘Request Elevator’ Use-Case of an ECS Application

Table 3.5: Event-Details of ECS Application

Event#	Event-Name	Event-Label
E1	user press elevator button	E1
E2	elevator button sensor reads the destination floor request and notifies it to system	E1,E2
E3	system update the request	E1,E2,E3
E4	WHILE request queue is not empty	E1,E2,E3,E4
E5	IF elevator is idle THEN	E1,E2,E3,E4(T),E5
E6	Determine direction	E1,E2,E3,E4(T),E5(T),E6
E7	system commands the door to close	E1,E2,E3,E4(T),E5(T),E6,E7
E8	IF door = closed THEN	E1,E2,E3,E4(T),E5(T),E6,E7,E8
E9	System commands to start the motor to move to the determined direction	E1,E2,E3,E4(T),E5(T),E6,E7,E8(T),E9
	ENDIF	{E1,E2,E3,E4(T),E5(T),E6,E7,E8(T),E9} OR {E1,E2,E3,E4(T),E5(T),E6,E7,E8(F)}
E10	As the elevator is approaching the floors, floor sensor detects the floor # and notifies the system	E1,E2,E3,E4(T),E5(F),E10
E11	IF elevator has to stop at that floor THEN	E1,E2,E3,E4(T),E5(F),E10,E11
E12	System commands the motor to stop	E1,E2,E3,E4(T),E5(F),E10,E11(T),E12
E13	IF motor= stopped THEN	E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13
E14	System commands the elevator door to open	E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14
E15	IF door=opened THEN	E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14,E15
E16	system updates the request queue	E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14,E15(T),E16

Event#	Event-Name	Event-Label
	ENDIF	{E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14,E15(T),E16} OR {E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14,E15(F)}
	ENDIF	{E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14,E15(T),E16} OR {E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14,E15(F)} OR {E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(F)}
	ENDIF	{E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14,E15(T),E16} OR {E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14,E15(F)} OR {E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(F)} OR E1,E2,E3,E4(T),E5(F),E10,E11(F)
	ENDIF	{E1,E2,E3,E4(T),E5(T),E6,E7,E8(T),E9} OR {E1,E2,E3,E4(T),E5(T),E6,E7,E8(F)} OR {E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14,E15(T),E16} OR {E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14,E15(F)} OR {E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(F)} OR {E1,E2,E3,E4(T),E5(F),E10,E11(F)}
	ENDWHILE	{E1,E2,E3,E4(T),E5(T),E6,E7,E8(T),E9} OR {E1,E2,E3,E4(T),E5(T),E6,E7,E8(F)} OR {E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14,E15(T),E16} OR {E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14,E15(F)} OR {E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(F)} OR {E1,E2,E3,E4(T),E5(F),E10,E11(F)} OR {E1,E2,E3,E4(F)}

Table 3.6: Event-Sequence Table for Scenario 1 of Elevator Control System Application

Event #	Precondition	Logical Time
E1		1
E2	E1	2
E3	E1,E2	3
E4	E1,E2,E3	4
E5	E1,E2,E3,E4(T)	5
E10	E1,E2,E3,E4(T),E5(F)	6
E11	E1,E2,E3,E4(T),E5(F),E10	7
E12	E1,E2,E3,E4(T),E5(F),E10,E11(T)	8
E13	E1,E2,E3,E4(T),E5(F),E10,E11(T),E12	9
E14	E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T)	10
E15	E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14	11
E16	E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14,E15(T)	12

Table 3.7: Event-Sequence Table for Scenario 2 of Elevator Control System Application

Event#	Precondition	Logical Time
E1		1
E2	E1	2
E3	E1,E2	3
E4	E1,E2,E3	4
E5	E1,E2,E3,E4(T)	5
E10	E1,E2,E3,E4(T),E5(F)	6
E11	E1,E2,E3,E4(T),E5(F),E10	7
E12	E1,E2,E3,E4(T),E5(F),E10,E11(T)	8
E13	E1,E2,E3,E4(T),E5(F),E10,E11(T),E12	9
E14	E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T)	10
E15	E1,E2,E3,E4(T),E5(F),E10,E11(T),E12,E13(T),E14	11

Table 3.8: Event-Sequence Table for Scenario 3 of Elevator Control System Application

Event#	Precondition	Logical Time
E1		1
E2	E1	2
E3	E1,E2	3
E4	E1,E2,E3	4
E5	E1,E2,E3,E4(T)	5
E10	E1,E2,E3,E4(T),E5(F)	6
E11	E1,E2,E3,E4(T),E5(F),E10	7
E12	E1,E2,E3,E4(T),E5(F),E10,E11(T)	8
E13	E1,E2,E3,E4(T),E5(F),E10,E11(T),E12	9

Table 3.9: Event-Sequence Table for Scenario 4 of Elevator Control System Application

Event#	Precondition	Logical Time
E1		1
E2	E1	2
E3	E1,E2	3
E4	E1,E2,E3	4
E5	E1,E2,E3,E4(T)	5
E6	E1,E2,E3,E4(T),E5(T)	6
E7	E1,E2,E3,E4(T),E5(T),E6	7
E8	E1,E2,E3,E4(T), E5(T),E6,E7	8
E9	E1,E2,E3,E4(T), E5(T),E6,E7	9

Table 3.10: Event-Sequence Table for Scenario 5 of Elevator Control System Application

Event#	Precondition	Logical Time
E1		1
E2	E1	2
E3	E1,E2	3
E4	E1,E2,E3	4
E5	E1,E2,E3,E4(T)	5
E6	E1,E2,E3,E4(T), E5(T)	6
E7	E1,E2,E3,E4(T), E5(T),E6	7
E8	E1,E2,E3,E4(T), E5(T),E6,E7	8

Table 3.11: Event-Sequence Table for Scenario 6 of Elevator Control System Application

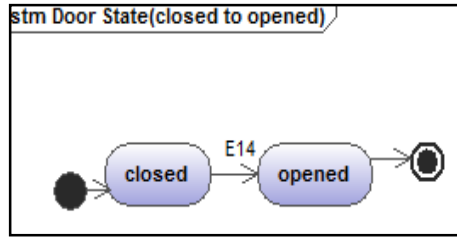
Event#	Precondition	Logical Time
E1		1
E2	E1	2
E3	E1,E2	3
E4	E1,E2,E3	4
E5	E1,E2,E3,E4(T)	5
E10	E1,E2,E3,E4(T),E5(F)	6
E11	E1,E2,E3,E4(T),E5(F),E10	7

Table 3.12: Event-Sequence Table for Scenario 7 of Elevator Control System Application

Event#	Precondition	Logical Time
E1		1
E2	E1	2
E3	E1,E2	3
E4	E1,E2,E3	4

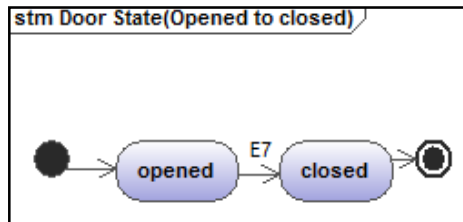
Step II: Identifying ‘Event-Sequence-State-Transitions’ for each Scenario

The participating components in the ‘*Request-Elevator*’ use-case functionality are ‘Motor’ and ‘Door’. The valid states of ‘Motor’ component are ‘stopped’ and ‘moving’. Similarly, the valid states of ‘Door’ component are ‘opened’ and ‘closed’. At the time of pressing of the floor button, the elevator is either in a stationary mode (i.e. positioned at some floor other than the requested floor) or in a servicing mode (i.e. serving any other user’s request). If the elevator is in stationary mode, then the states of both ‘Motor’ and ‘Door’ components are ‘stopped’ and ‘opened’ respectively. Otherwise, (if the elevator is in servicing mode) the states of ‘Motor’ and ‘Door’ components are ‘moving’ and ‘closed’ respectively. In all the scenarios where the execution of the conditional event E5 (‘IF elevator is idle THEN’) returns true, the states of components ‘Motor’ and ‘Door’ components are ‘stopped’ and ‘opened’ respectively. Otherwise, the state of the components ‘Motor’ and ‘Door’ are ‘moving’ and ‘closed’ respectively. So, two sets of state diagrams, each for Motor and Door components are drawn in this case study application. The drawn state diagrams for both the ‘Door’ and ‘Motor’ components are shown in Figure 3.12 and Figure 3.13 respectively. There are drawn two state diagrams for ‘Motor’ and ‘Door’ components.



(a) Door State Diagram when the initial state is 'closed'

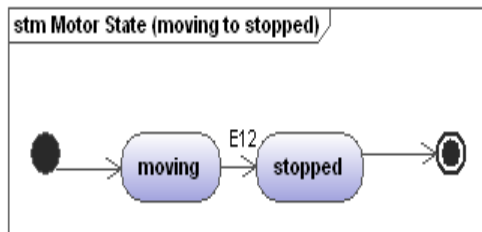
Event E14 ('System commands the elevator door to open') causes the state of the Door component be changed to 'opened'



(b) Door State Diagram when the initial state is 'opened'

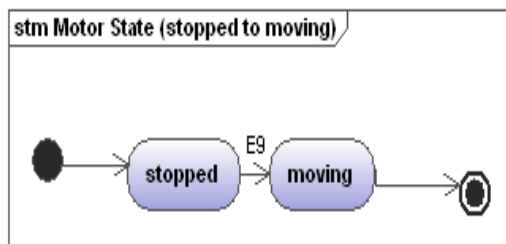
Event E7 (System commands the elevator door to close) causes the state of the Door component be changed to 'closed'

Figure 3.12: Door State Diagrams for Elevator Control System Application



(a) Motor state Diagram when the initial state is 'moving'

Event E12 (System commands the motor to stop) causes Motor state to be changed to 'stopped'



(b) Motor state Diagram when the initial state is 'stopped'

Event E9 (System commands to start the motor to move to the determined direction) causes Motor state to be changed to 'moving'

Figure 3.13: Motor State Diagrams for Elevator Control System Application

The output of Step II results in seven ‘*Event-Sequence-State-Transition*’ tables (one table for each scenario) and these tables for are shown in Table 3.13 to Table 3.19. The ‘*Event-Sequence*’ tables as shown in Tables 3.6 to Table 3.12 and UML state diagrams as shown in Figure 3.12 and Figure 3.13 are supplied as inputs to this step.

Table 3.13: Event-Sequence-State-Transition Table for Scenario 1

Event#	Door	Motor
E1	closed	moving
E2	closed	moving
E3	closed	moving
E4	closed	moving
E5	closed	moving
E10	closed	moving
E11	closed	moving
E12	closed	stopped
E13	closed	stopped
E14	opened	stopped
E15	opened	stopped
E16	opened	stopped

Table 3.14: Event-Sequence-State-Transition Table for Scenario 2

Event#	Door	Motor
E1	closed	moving
E2	closed	moving
E3	closed	moving
E4	closed	moving
E5	closed	moving
E10	closed	moving
E11	closed	moving
E12	closed	stopped
E13	closed	stopped
E14	opened	stopped
E15	opened	stopped

Table 3.15: Event-Sequence-State-Transition Table for Scenario 3

Event#	Door	Motor
E1	closed	moving
E2	closed	moving
E3	closed	moving
E4	closed	moving
E5	closed	moving
E10	closed	moving
E11	closed	moving
E12	closed	stopped
E13	closed	stopped

Table 3.16: Event-Sequence-State-Transition Table for Scenario 4

Event#	Door	Motor
E1	opened	stopped
E2	opened	stopped
E3	opened	stopped
E4	opened	stopped
E5	opened	stopped
E6	opened	stopped
E7	closed	stopped
E8	closed	stopped

Table 3.17: Event-Sequence-State-Transition Table for Scenario 5

Event#	Door	Motor
E1	opened	stopped
E2	opened	stopped
E3	opened	stopped
E4	opened	stopped
E5	opened	stopped
E6	opened	stopped
E7	closed	stopped
E8	closed	stopped
E9	closed	moving

Table 3.18: Event-Sequence-State-Transition Table for Scenario 6

Event#	Door	Motor
E1	closed	moving
E2	closed	moving
E3	closed	moving
E4	closed	moving
E5	closed	moving
E10	closed	moving
E11	closed	moving

Table 3.19: Event-Sequence-State-Transition Table for Scenario 7

Event#	Door	Motor
E1	closed	moving
E2	closed	moving
E3	closed	moving
E4	closed	moving

Step III: Identifying ‘State-Transition-Errors’

The execution of Step III results in the instantiation of ‘State-Transition-Errors’ as tabulated in Table 3.20.

Table 3.20: State-Transition-Error Table for Elevator Control Application

Error#	Error_Name	Event#	Effect
ER1	Motor Fails to Stop	E12	Motor != stopped
ER2	System Fails to Give Stop Motor Command	E12	Motor != stopped
ER3	Door Fails to Open	E14	Door != opened
ER4	System Fails to Give Door Open Command	E14	Door != opened
ER5	Door Fails to Close	E7	Door != closed
ER6	System Fails to Give Door Close Command	E7	Door != closed
ER7	Motor Fails to Move	E9	Motor != moving
ER8	System Fails to Give Move Motor Command	E9	Motor != moving

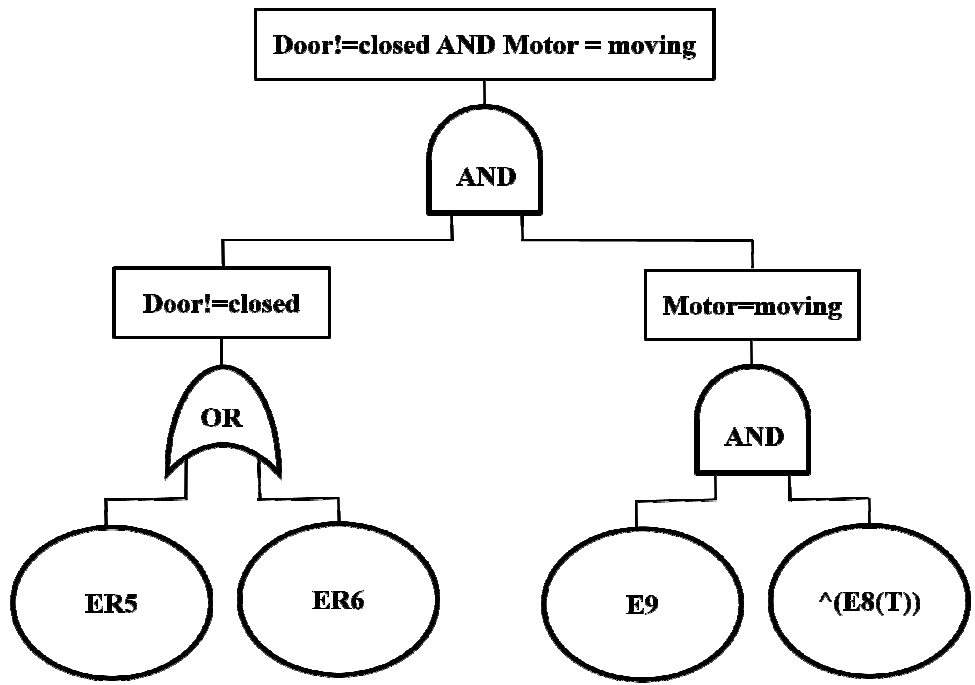
Step IV: Construct Fault Trees

Two Type 1 and one Type 2 composite hazardous-states are considered for this case study application. The Type 1 composite hazardous-state(s)are: (i) ‘Door!= closed AND Motor = moving’ and (ii) ‘Motor!= stopped AND Door = opened’. The Type 2 composite hazardous-state is: ‘Door = opened AND Motor = moving’.

The fault tree XML file generated for the Type 1 composite hazardous-state ‘Door!= closed AND Motor = moving’ is shown in Figure 3.14 and fault tree constructed from this XML file is shown in Figure 3.15.

```
<?xml version="1.0" encoding="UTF-8"?>
<Fault-Tree><Intermediate-Event><Title>Door!=closed.AND.Motor=moving</Title>
<And-Gate><Intermediate-Event><Title>Door!=closed</Title><And-Gate><Basic-Event><Title>ER5</Title></Basic-Event><Basic-Event><Title>ER6</Title></Basic-Event></And-Gate></Intermediate-Event><Intermediate-Event><Title>Motor=moving</Title><And-Gate><Basic-Event><Title>E9</Title></Basic-Event><Basic-Event><Title>^(E8(T))</Title></Basic-Event></And-Gate></Intermediate-Event></And-Gate></Intermediate-Event></Fault-Tree>
```

Figure 3.14: faulttree.xml file for Hazardous-State: Door!=closed AND Motor=moving



- ER5: Door Fails to Close
- ER6: System Fails to Give Door Close Command
- E9: System Commands to Start the Motor to Move to the Determined Direction
- ^{(E8(T))}: E8 is Conditional Event and is Wrongly Evaluated as True

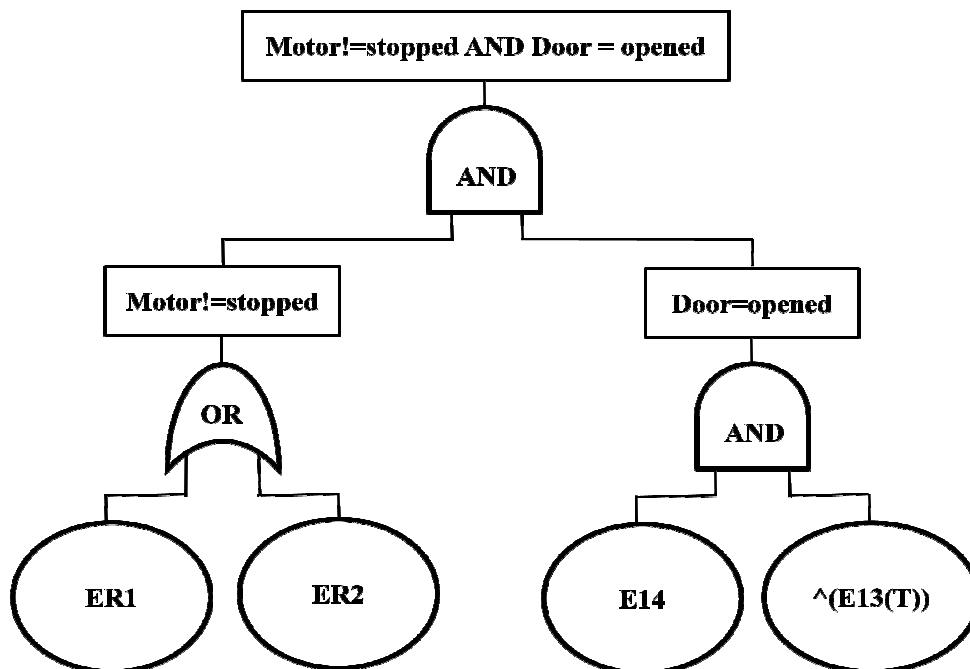
Figure 3.15: Fault Tree for Hazardous State Door!=closed AND Motor = moving

The basic error events corresponding to the conditional events in a fault tree are represented using the form ‘ $\wedge(C(T))$ ’ where ‘C’ is a conditional type event. It represents the error situation where ‘C’ is wrongly evaluated as true. For example, in Figure 3.15, the error event ‘ $\wedge(E8(T))$ ’ indicates that the event ‘E8’ is a conditional event and is wrongly evaluated as true.

The generated fault tree XML file for Type I composite hazardous-state ‘*Motor!=stopped AND Door = opened*’ is shown in Figure 3.16 and a fault tree constructed from this file is shown in Figure 3.17.

```
<?xml version="1.0" encoding="UTF-8"?>
<Fault-Tree><Intermediate-Event><Title>Motor!=stopped.AND.Door=opened</Title>
<And-Gate><Intermediate-Event><Title>Motor!=stopped</Title>
<And-Gate><Basic-Event><Title>ER1</Title></Basic-Event><Basic-
Event><Title>ER2</Title></Basic-Event></And-Gate></Intermediate-
Event><Intermediate-Event><Title>Door=opened</Title><And-Gate><Basic-
Event><Title>E14</Title></Basic-Event><Basic-Event><Title> $\wedge(E13(T))$ </Title></Basic-
Event></And-Gate></Intermediate-Event></And-Gate></Intermediate-Event></Fault-
Tree>
```

Figure 3.16: faulttree.xml file for Hazardous-State: Motor !=stopped AND Door = opened



- ER1: Motor Fails to Stop
- ER2: System Fails to Give Stop Motor Command
- E14: System Commands the Elevator Door to Open
- $\wedge(E13(T))$: E13 is Conditional Event and is Wrongly Evaluated as True

Figure 3.17: Fault Tree for Hazardous-State: Motor !=stopped AND Door = opened

The fault tree XML file generated for Type 2 Composite hazardous-state selected ‘Door = opened AND Motor = moving’ is shown in Figure 3.18. The constructed fault tree from this XML file is shown in Figure 3.19.

```
<?xml version="1.0" encoding="UTF-8"?>
<Fault-Tree><Intermediate-Event><Title>Motor!=stopped.AND.Door=opened</Title>
<And-Gate><Intermediate-Event><Title>Motor!=stopped</Title>
<And-Gate><Basic-Event><Title>ER1</Title></Basic-Event><Basic-
Event><Title>ER2</Title></Basic-Event></And-Gate></Intermediate-Event><Intermediate-
Event><Title>Door=opened</Title><And-Gate><Basic-Event><Title>E14</Title></Basic-
Event><Basic-Event><Title>^(E13(T))</Title></Basic-Event></And-Gate></Intermediate-
Event></And-Gate></Intermediate-Event></Fault-Tree>
```

Figure 3.18: faulttree.xml file for Hazardous-State: Door=opened AND Motor = moving

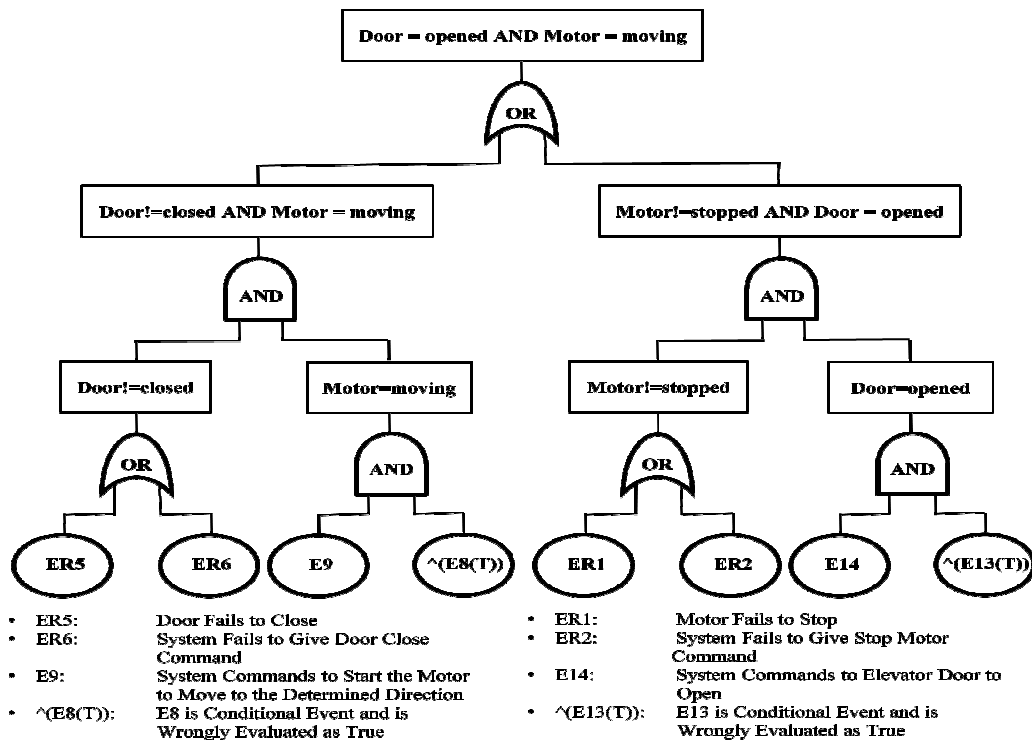


Figure 3.19: Fault Tree for Hazardous-State Door=opened AND Motor = moving

3.7 MOTIVATING EXAMPLE 2:RAILWAY TRACK DOOR CONTROL SYSTEM APPLICATION

The second example is about safety-critical Railway Track Door Control system for avoiding the accidents at the railway crossing. The events of this use-case are executed whenever the rail track door is to be closed in response to an interrupt

received from the track sensors attached to the rail tracks. The interrupt from the rail track sensors informs the arrival of the train. The use-case description file for this case study is shown in Figure 3.20.

```

/*****Basic Details*****/
/*      Use Case      :  Opening of Railway Track_Door */
/*      Initiating Actor :  Rail Track Controller      */
/*****/
/*      Event Details      */
rail track sensors detect the arrival of train and interrupts the rail track control system
upon interruption by track sensors, rail track control system instructs track_door to be
closed
IF track_door = closed THEN
rail track control system instructs track_signal to go green
rail track control system waits for the next interrupt from the track sensor
ELSE
report track_door_failure
ENDIF
    
```

Figure 3.20: Use Case Description File for ‘Open Rail Track Door’ Use-Case of RTCS Application

Step I: Extract ‘Event-Sequence’ for Each Scenario

The use-case description file of Figure 3.20 is used as an input in this step. The description of various ‘Event-Details’ is shown in Table 3.21.

Table 3.21: Event-Details of RTCS Application

Event#	Event-Name	Event-Label
E1	rail track sensors detect the arrival of train and interrupts the rail track control system	E1
E2	upon interruption by track sensors, rail track control system instructs track_door to be closed	E1,E2
E3	IF track_door = closed THEN	E1,E2,E3
E4	rail track control system instructs track_signal to go green	E1,E2,E3(T),E4
E5	rail track control system waits for the next interrupt from the track sensor	E1,E2,E3(T),E4,E5
E6	report track_door_failure	E1,E2,E3(F),E6
	ENDIF	{E1,E2,E3(T),E4,E5} OR E1,E2,E3(F),E6

As per the 'Event-Label' value of the 'ENDIF', there are two possible scenarios in this example. The *Event-Sequence* tables generated for the Scenario 1 and Scenario 2 of this application are shown in Table 3.22 and Table 3.23 respectively.

Table 3.22: Event-Sequence table for Scenario 1

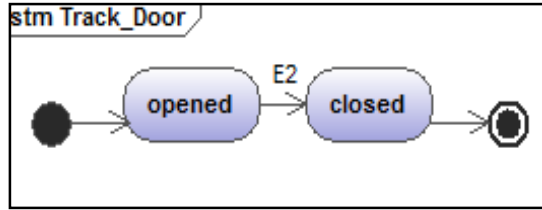
Event#	Precondition	Logical Time
E1		1
E2	E1	2
E3	E1,E2	3
E4	E1,E2,E3(T)	4
E5	E1,E2,E3(T),E4	5

Table 3.23: Event-Sequence table for Scenario 2

Event#	Precondition	Logical Time
E1		1
E2	E1	2
E3	E1,E2	3
E6	E1,E2,E3(F)	4

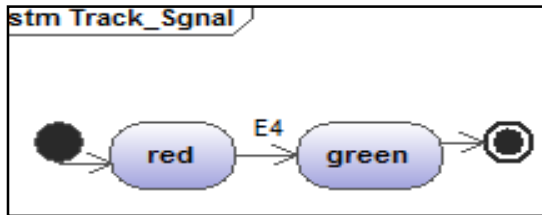
Step II: Identify 'Event-Sequence-State-Transitions' for Each Scenario

The participating components in this case study are 'Track_Door' and 'Track_Signal'. The valid states of the 'Track_Door' component are 'opened' and 'closed'. The valid states of the 'Track_Signal' component are 'red' and 'green'. The state diagrams of the 'Track_Door' and 'Track_Signal' components are shown in Figure 3.21. The initial state of the 'Track_Door' component is 'opened' whereas the initial state of the 'Track_Signal' component is 'red'. The 'Track_Door' component changes its state from 'opened' to 'closed' during the execution of event E2 (rail track control system instructs track_door to be closed). Similarly, the 'Track_Signal' component changes its state from 'red' to 'green' during the execution of event E4 (rail track control system instructs track_signal to go green). The state transition pattern of both 'Track_Door' and 'Track_Signal' components is same for both the scenarios. That is why only one UML state diagram is drawn for both 'Track_Door' and 'Track_Signal' components.



Event E2 → rail track control system instructs track_door to be closed

(a) Track_Door State Diagram



Event E4 → rail track control system instructs track_signal to go green

(b) Track_Signal State Diagram

Figure 3.21: Input State Diagrams for Rail Track Door Control System Application

The ‘Event-Sequence-State-Transition’ tables for Scenario 1 and Scenario 2 are shown in Table 3.24 and Table 3.25 respectively.

Table 3.24: Event-Sequence-State-Transition table for Scenario 1

Event	Track_Door	Track_Signal
E1	opened	red
E2	closed	red
E3	closed	red
E4	closed	green
E5	closed	green

Table 3.25: Event-Sequence-State-Transition table for Scenario 2

Event	Track_Door	Track_Signal
E1	opened	red
E2	closed	red
E3	closed	red
E6	closed	red

Step III: Identify ‘State-Transition-Errors’

The ‘*State-Transition-Error*’ table created for this example is shown in Table 3.26. There are two state transition events E2 and E4 and that’s why there are four state errors with error numbers as ER1, ER2, ER3 and ER4.

Table 3.26: State-Transition-Error Table

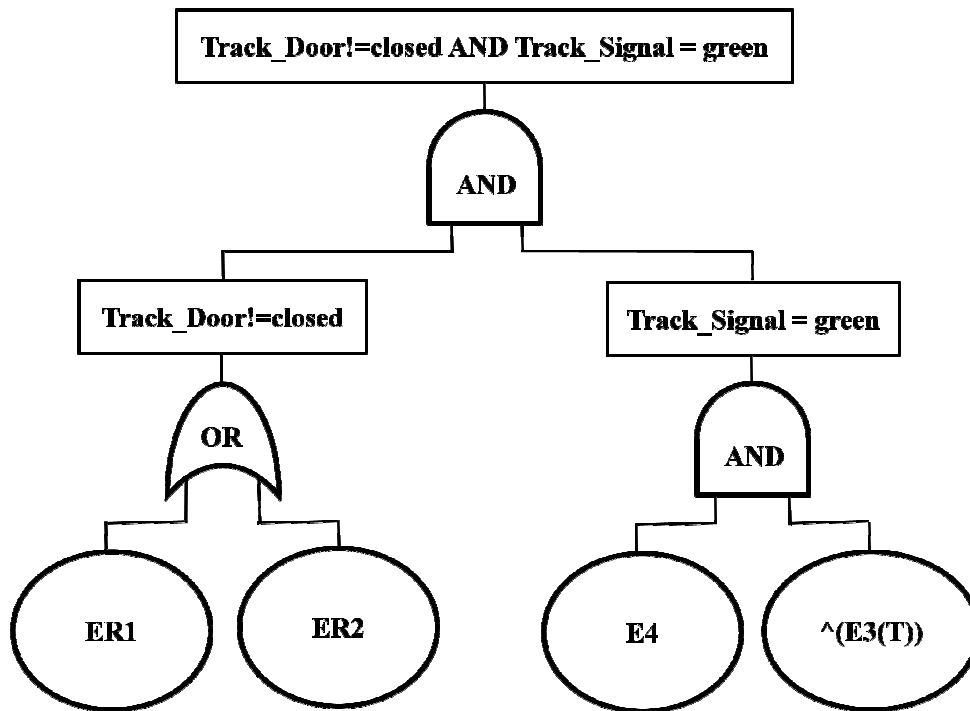
Error#	Error_Name	Event#	Effect
ER1	RTCS System Fails to Give Door Close Command	E2	Track_Door != closed
ER2	Track_Door Failure	E2	Track_Door != closed
ER3	RTCS System Fails to Give Green Signal Command	E4	Track_Signal != green
ER4	Track_Signal Failure	E4	Track_Signal != green

Step IV: Construct Fault Trees

There is considered only one Type 1 composite hazardous-state which is ‘*Track_Door!= closed AND Track_Signal =green*’ (i.e. *The truck door has not closed, but the track signal has gone green*). The generated fault tree XML file for this hazardous-state is shown in Figure 3.22. The fault tree generated from this input file is shown in Figure 3.23.

```
<?xml version="1.0" encoding="utf-8"?><Fault-Tree><Intermediate-Event><Title>Track_Door!=closed.AND.Track_Signal=green</Title><And-Gate><Intermediate-Event><Title>Track_Door!=closed</Title><And-Gate><Basic-Event><Title>ER1</Title></Basic-Event><Basic-Event><Title>ER2</Title></Basic-Event></And-Gate></Intermediate-Event><Intermediate-Event><Title>Track_Signal=green</Title><And-Gate><Basic-Event><Title>E4</Title></Basic-Event><Basic-Event><Title>^(E3(T))</Title></Basic-Event></And-Gate></Intermediate-Event></And-Gate></Intermediate-Event></Fault-Tree>
```

Figure 3.22: faulttree.xml file for Hazardous-State: Track_Door !=closed AND Track_Signal = green



- ER1: RTCS System Fails to Give Door Close Command
- ER2: Track_Door Failure
- E4: RTCS System Instructs the Track_Signal to go Green
- $^{(E3(T))}$: E3 is a Conditional Event and is Wrongly Evaluated as True

Figure 3.23: Fault Tree Generated For Hazardous-State Track_Door !=closed AND Track_Signal = green

The snapshots of the fault trees constructed using a FaultCAT tool, from the faulttree.xml files as shown in Figure 3.12, Figure 3.14, Figure 3.18 and Figure 3.22, are also shown in Appendix-I, Appendix-II, Appendix-III and Appendix-IV, respectively.

3.8 VALIDATION OF THE ALGORITHM

The fault trees constructed by the presented approach for the hazardous-state(s) of an Elevator Control System (ECS) application are compared against the manually constructed fault trees for the same hazardous-states (Vyas and Mittal, 2012) and the results are found to be of similar quality.

Are the Fault Trees Constructed by the Presented SFTA Approach are Correct?

Schellhorn et al. (Schellhorn et al,2002) states that the correctness condition of fault trees guarantees that if the cause happens, the consequence must happen too and the consequence must not happen without the cause. This correctness condition can be interpreted as follows:

A fault tree ('FT') constructed for a hazardous-state ('X') is considered as correct if 'FT' does not contain a single basic erroneous event 'E' that actually does not contribute to the occurrence of hazardous-state 'X'.

The presented approach constructs a software fault tree only for state level hazards. Each state level term in the hazardous-state either uses a negation (such as $X \neq x$) or true (such as $Y = y$) symbol with the restriction that the first component should use a negation symbol. If the constructed fault tree for a hazardous-state ('X') contains a basic erroneous event 'E' that actually does not contribute towards the occurrence of 'X' then 'E' is either an invalid state level error selected from the '*State-Transition-Error*' table of Step III (for negation \neq symbol) or an invalid event from '*Event-Sequence*' tables of Step I (for true $=$ symbol). This can only happen because of any one of the following reasons.

- (i) The supplied use-case description file may be incorrect, or
- (ii) An erroneous state transition event is selected in Step III. It can happen if and only if the state diagrams supplied as inputs in Step II are incorrect.

But, the proposed approach operates with the assumption that use-case description file and UML state diagrams are correctness of the basic inputs.

Are the Fault Trees Constructed by the Proposed SFTA Approach are Complete?

The completeness issue deals with the coverage of errors. Schellhorn et al. (Schellhorn, 2002) states that the completeness condition of fault trees guarantees that all causes have been listed. This completeness condition can be interpreted as follows:

A fault tree ('FT') constructed for a hazardous-state ('X') is considered to be complete if it contains every basic erroneous event 'E' that contributes to the occurrence of the hazardous-state 'X'.

The presented approach guarantees the coverage of software-related errors provided the pseudo code description of the use-case functionality is complete. But the approach considered only one error for the components ('Door', 'Motor'). But in actual situations, there can be multiple reasons for the failure of any device/component. For example, the

door failure may happen either because of ‘Electrical short Circuit’ or because of ‘DoorSensor Failure’ etc. So the events that specifically belong to a device-failure category are to be expanded further in order to complete the fault tree.

3.9 COMPARATIVE ANALYSIS

The past applications of the SFTA approach in use-case based requirements analysis phase are mostly manual and time-consuming (Balz and Goll 2005, Douglass 2009, Gupta et al 2012, Tiwari et al 2012). SFTA approach for use-cases as reported by Tiwari (Tiwari et al, 2012) first converts the given formal use case realization template (UCRT) into a tree known as *success tree* and then converts the *success tree* into its corresponding fault tree by complementing the nodes of the *success tree* and the fault tree construction process is manual. The whole use-case functionality is converted into a single *success tree*. In general, a fault tree construction process is hazard specific and multiple hazards can occur during the realization of single use-case functionality. Moreover, the selected hazard-state can occur in multiple scenarios of the same use-case functionality also.

The presented SFTA approach is automatic and is algorithmically very simple. It has following advantages.

- (i) The technique is automated but only for constructing fault trees for state level hazards.
- (ii) The approach can handle the use-case description file of any size.
- (iii) The approach is easily scalable to any number of state variables.

The main shortcoming(s) of the proposed approach are as follows.

- (i) The software fault tree is constructed only for state level hazards. There are some hazardous situations that cannot be fully expressed via state level hazards such as speed of elevator increases suddenly (for ECS application), incorrect result of some computation etc.
- (ii) The proposed approach in the present form cannot handle the case where the participating components are experiencing concurrent state transitions.
- (iii) The approach takes into account only state related errors (i.e. the errors that occur only during state transition events). The effects of the errors that occur at events other than the state transition events (i.e. the events where no component is changing its state) have not been analyzed by the approach.

Software Failure Modes and Effects Analysis Approach in Use-Case Based Requirements Analysis Phase

The efforts to automate or semi-automate the application of the Software Failure Modes and Effects Analysis (SFMEA) approach in use-case based requirements analysis phase have not been successful so far. This chapter describes the developed semi-automated SFMEA technique in use-case based requirements analysis phase. The main weakness of the SFTA approach as discussed in Chapter 3 is that it only considers the event-related errors occurring at the state transition events (events that cause changes in the state of a component). The developed SFMEA approach overcomes this drawback by considering all the event-related errors. The approach is applied on two safety-critical case study applications, namely Rail Track Door Control System (RTCS) application discussed in Chapter 3 and Insulin Delivery System (IDS) (Sommerville, 2005). The formal textual description of a given use-case functionality and the UML state diagrams drawn for the participating components are used as the inputs in this proposed approach. The approach first identifies all the event-related errors that can occur in the system and then investigates the critical effects of these errors on the system.

4.1 PURPOSE OF THE PROPOSED SFMEA APPROACH

Like SFTA, the available literature about the application of the SFMEA approach in use-case based requirements analysis phase is also manual and time-consuming. Wentao and Hong (Wentao and Hong, 2009) used manual SFMEA approach on the use-case model of an Automated Teller Machine (ATM). Troubitsyna (Troubitsyna, 2011) applied manual application of the SFMEA approach on the use-case model of an autonomous robot by defining an auxiliary use-case corresponding to each use-case functionality. Nggada (Nggada, 2012) applied SFMEA approach on the use-case model of brake by wire system (BBS). Gupta (Gupta et al, 2012) and Tiwari (Tiwari et al, 2012) applied the manual application of SFMEA approaches in use-case based requirements analysis phase by taking the formal textual descriptions of a use-case as an input.

The proposed approach integrates and semi-automates the application of the SFMEA approaches in use-case based requirements analysis process. The approach is forward in nature in the sense that it investigates the state level effects caused by various event-related errors in the system.

4.2 ASSUMPTIONS FOR THE PROPOSED SFMEA APPROACH

The assumptions of the proposed SFMEA approach are identical to the assumptions made in the SFTA approach Chapter 3 (Section 3.2). In addition, while investigating the effects of any event-related error, the approach also assumes that *no error has occurred in the system before the execution of the selected event, i.e. the effects are analyzed only for one event-related error at a time.*

4.3 OVERVIEW OF THE PROPOSED SFMEA APPROACH

There are four steps in the proposed approach and an overview of each step is given below.

The working logic of the first and second step of the proposed SFMEA approach is identical to Step I and Step II of the SFTA approach discussed in Chapter 3. The structures of both ‘Event-Details’ and ‘Event-Sequences’ and the ‘Event-Sequence-State-transitions’ used in the approach are also identical to the SFTA approach discussed in Chapter 3. Like SFTA approach of Chapter 3, the state diagrams in the proposed SFMEA approach are also accepted in machine readable format i.e. XMI (XML Metadata Interchange) format. The Altova UML (Altova-UModel, 2014) tool is used to draw the required state diagrams and each state diagram is exported to XMI format using the same tool.

The third step takes the ‘Event-Details’ of various events extracted in the first step and the ‘Event-Sequence-State-Transitions’ of various scenarios identified in the second step and identifies the various event-related errors corresponding to each executable event. The attributes of the identified errors are stored in a tabular form that has the structure as shown in Table 4.1.

Table 4.1: Structure of Event-Errors

Event#	Error#	Error-Description	Type
<<Event# where an error can occur>>	<<A unique Error Number assigned to each error>>	<<description of the error>>	<<The type is 1 for stop-type error and 2 for propagating-type error>>

The fourth step investigates the effects of various event-related errors identified in the third step. The ‘Event-Details’ extracted in the first step, the ‘Event-Sequence-State-Transition’ identified in the second step and the ‘Events-Errors’ identified in the third step are used as the inputs. The effects of the event-related errors are stored separately for each scenario in a tabular form known as ‘Events-Errors-Effects-Analysis’. The structure of the ‘Events-Errors-Effects-Analysis’ has three main fields, namely (i) Event#, (ii) Error# and (iii) Effects as shown in Table 4.2. The ‘Effects’ column is further sub-divided into various event sub-columns and the number of these event sub-columns depend upon the number of events in the associated scenario.

Table 4.2: Structure of ‘Event-Errors-Effects-Analysis’

Event#	Error#	Effects			
		E1	E2	...	En
<<Event Number where an error has occurred>>	<<Error number that has occurred at the event>>	<<Effects of the errors on various executable events>>			

An overview of the four steps of the proposed SFMEA approach is shown in Figure 4.1.

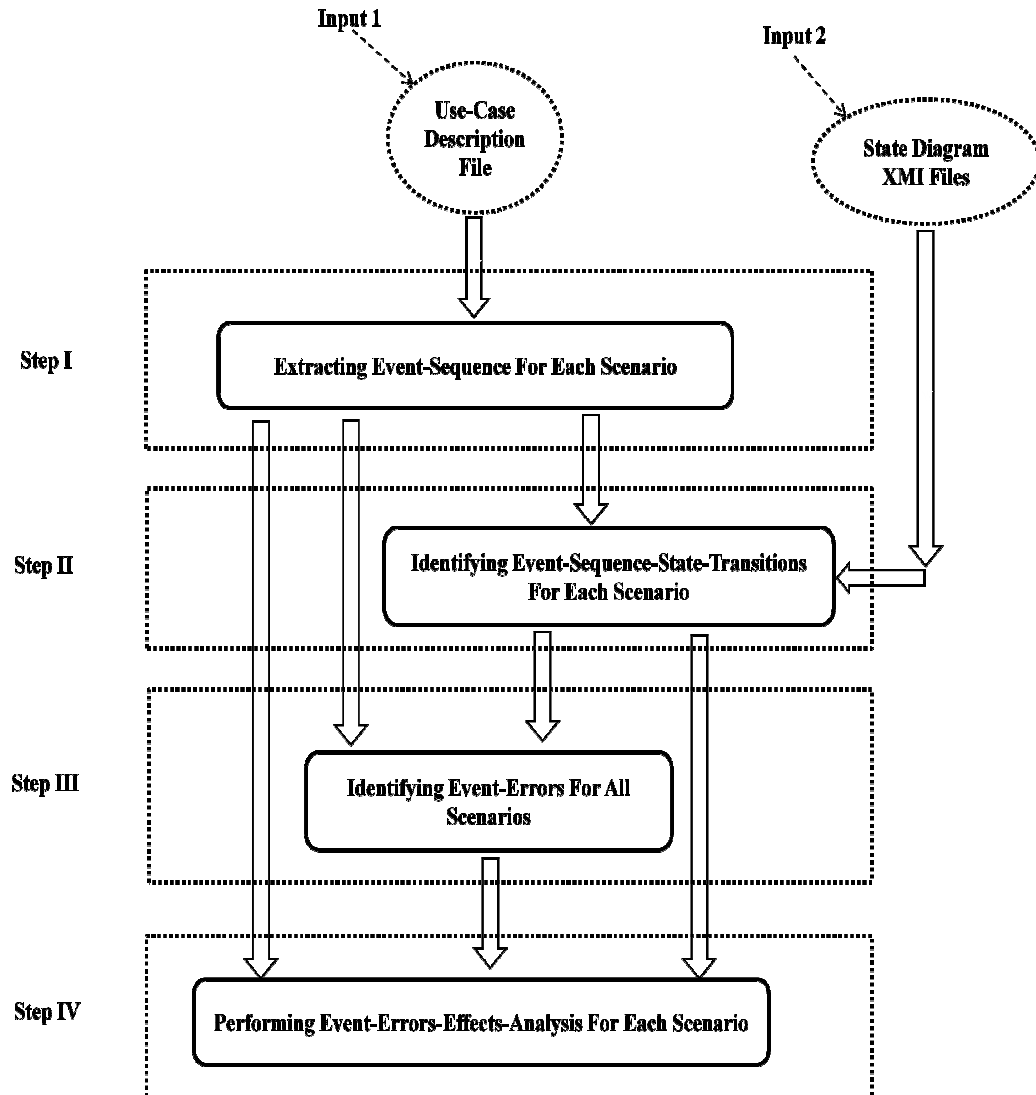


Figure 4.1: Overview of the Proposed SFMEA approach

4.4 SFMEA ALGORITHM

The proposed SFMEA algorithm is explained in detail in the following sections. The use-case model and use-case description file, as shown in Figure 3.2 and Figure 3.3 in Section 3.5 of Chapter 3, are used to explain the logic.

4.4.1 Step I: Extracting Event-Sequences for each Scenario

This step is identical to the Step I of the SFTA approach discussed in Chapter 3. The outputs of this step on the selected use-case description file are shown in Figure 3.4 and Figure 3.5 in Chapter 3.

4.4.2 Step II: Identifying Event-Sequence-State-Transitions For Each Scenario

This step is identical to the Step II of the SFTA approach discussed in Chapter 3. The output of this step on the selected example is shown in Figure 3.8 in Chapter 3.

4.4.3 Step III: Identifying ‘Event-Errors’ for all Scenarios

This step identifies the event-related errors that can occur during the execution of the events and records these errors in a tabular form as shown in Table 4.1. Two types of event-related errors are considered for the SFMEA approach and these are explained below.

- (i) *Propagating Errors*: These types of errors affect the execution of all the successive events in the scenario. These errors do not prevent or stop the execution of successive events.

How to identify Propagating Errors?

The system can experience these types of errors during the execution of both normal and conditional types of events. For conditional types (True/False) of events, the system can experience two types of propagating errors and these types are explained below with an example.

Suppose the ‘C’ is a conditional type of event. The first type of propagating error ‘Er’ occurs when the actual value of ‘C’ is ‘false’, but during execution, it is evaluated as ‘true’. The second type of propagating error ‘Es’ occurs when the actual value of ‘C’ is ‘true’, but during execution, it is evaluated as ‘false’.

The propagating errors that occur during the execution of the normal events are to be identified manually.

- (ii) *Stop Errors*: These types of event-related errors prevent the successive events from execution, i.e. successive events are not executed at all and the execution stops at the selected event.

How to Identify Stop Errors?

For a conditional type of events there is considered only one type of stop error and it represents the situation where the conditional events fail to execute at all and the execution of all the successive events is suspended.

For normal types of events, the system can experience different types of stop errors and these are explained below with an example.

Suppose ‘N’ is a normal event. If the execution of ‘N’ results in the state transition of any component, then two states-related errors (both are considered stop type errors), identical to the state-related errors considered in the Step III of the SFTA approach of Chapter 3, are identified corresponding to event ‘N’. If the execution of ‘N’ does not change the state of any component, then there is considered only one stop type error which represents the situation where the events ‘N’ fails to execute at all and the execution of all the successive events is suspended.

The pseudo code of this step in the form of a procedure named ‘identify-Event-Related-Errors()’ is given below.

<i>Procedure</i>	<i>Identify-Event-Related-Errors()</i>
<i>Input(s)</i>	<i>Event-Details Table of Step I and Event-Sequence-State-Transitions</i>
<i>Output</i>	<i>Event-Errors</i>

```

FOR each executable event in Event-Details
    IF event is a conditional type event THEN
        define one stop-type error for the event;
        define two propagating-type errors for the event;
    ENDIF
    IF event is a normal type event THEN
        IF event is a changing the state of any component THEN
            define two stop-type state-related errors for the event;
        ELSE
            define one stop-type error for the event;
        ENDIF
    ENDIF
ENDFOR
    
```

If the ‘Event-Details’ as shown Figure 3.4 in Chapter 3 and the ‘Event-Sequence-State-Transitions’ as shown in Figure 3.8 in Chapter 3 are used as inputs in this step, then the output of this step results in the identification of ‘Event-Errors’ as shown in Table 4.3.

Table 4.3: Event-Errors for Scenario 1 and Scenario 2

Event#	Error#	Type	Error-Description
E1	ER1	1	Event E1 fails to execute
E2	ER2	1	Event E2 fails to execute
	ER3	2	Event E2 does a wrong computation
E3	ER4	1	Event E3 fails to execute at all
	ER5	2	Event E3 is True, but evaluated as False
	ER6	2	Event E3 is false, but evaluated as True
E4	ER7	1	Event E4 fails to execute at all
	ER8	1	State Error in component X
E5	ER9	1	Event E5 fails to execute at all
	ER10	1	State Error in component Y
E6	ER11	1	Event E6 fails to execute at all
E7	ER12	1	Event E7 fails to execute at all

All event-related errors in Table 4.3 are identified automatically, except the error ‘ER3’ at event ‘E2’. A stop type error where an event fails to execute and stops the execution of successive events is automatically identified for each event (for example the errors ER1, ER2, ER4, ER7, ER9, ER11 and ER12). The error ‘ER3’ is a propagating type error that occurs at normal type event ‘E2’ (Figure 3.4 in Chapter 3) and that is why it is identified manually. The event ‘E3’ is a conditional type event (Figure 3.4 in Chapter 3) and that is why two propagating type errors ‘ER5’ and ‘ER6’ and one stop type error ‘ER4’ are automatically identified for event ‘E3’. The execution of the event ‘E4’ results in the state transition of component ‘X’ (Figure 3.8 in Chapter 3) and that is why two stop type errors (‘ER7’ and ‘ER8’) are identified for this event. Similarly, the execution of the event ‘E5’ results in the state transition of component ‘Y’ (Figure 3.8 in Chapter 3) and that is why two stop type errors (‘ER9’ and ‘ER10’) are automatically identified for this event.

4.4.4 Step IV: Performing ‘Event-Errors-Effects-Analysis’ of each Scenario

This step performs the ‘*Event-Errors-Effects-Analysis*’ of each scenario. The algorithm investigates the effects of the propagating and stop types of errors as follows.

(a) *Investigating the effects of Stop-type errors*

All the state transitions that are supposed to occur during the successive executable events do not occur in the system. For example, consider the error ‘ER1’ of Table 4.3. This error prevents the execution of all the successive events E2,E3,E4,E5,E6,E7 in scenario 1 (see Figure 3.5 in Chapter 3). Hence all the expected state transitions such as $X=x_2$ at event ‘E4’ and $Y=y_2$ at event ‘E5’ are not observed in the system (see Figure 3.8 in Chapter 3). These effects are indicated using ‘!=’ symbol as $X \neq x_2$ and $Y \neq y_2$ under the respective event sub-columns in ‘Event-Errors-Effects-Analysis’ of the scenario.

(b) *Investigating the effects of Propagating-type errors*

These types of errors do not prevent the execution of the successive events and hence their effects are transmitted in the state transitions occurring during successive executable events. Consider error ‘ER3’ at event ‘E2’ in Table 4.3. In this situation, the state transitions that are occurring at events ‘E4’ and ‘E5’ are known as erroneous state transitions because these transitions are taking place under error conditions and are represented using the upper caret (‘^=’) symbol as $X \hat{=} x_2$ (i.e. The component X is erroneously changing its state) under the respective event sub-columns.

The pseudo code of this step in the form of a procedure named ‘perform-Event-Errors-Effects-Analysis’ is given below.

<i>Procedure</i>	<i>perform-Event-Errors-Effects-Analysis()</i>
<i>Input(s)</i>	<i>The outputs of the previous steps</i>
<i>Output(s)</i>	<i>Event-Errors-Effects-Analysis of Each Scenario</i>

FOR each Event-Sequence-State-Transition

 create a associated Event-Errors-Effects-Analysis;

FOR each event of Event-Sequence-State-Transition

Case: event is Normal event

FOR each error corresponding to the event

IF error is a stop-type **THEN**

Mark all the successive state transitions using '!=';

ELSE

Mark the successive state transitions using '^=';

ENDIF

ENDFOR

Case: event is Conditional event

Select the error which affects the Event-Sequence-State-Transition;

Mark the successive state transitions using '^=';

ENDFOR

ENDFOR

The effects of all the event-related errors for scenario 1, as shown in Table 4.3, are shown in Table 4.4.

The error 'ER1' (stop type error) prevents the state transitions from occurring at events 'E4' and 'E5'. Because no state transition is taking place during the early events E1, E2 and E3, the effects of the error 'ER1' are shown only under event sub-columns 'E4' and 'E5' respectively.

The error 'ER3' is a propagating type error and that's why the state transitions at events 'E4' and 'E5' are indicated as erroneous state transitions using '^=' symbol.

The error 'ER5' prevents the execution of scenario 1 (i.e. scenario 2 gets erroneously executed because of this error) and that's why the row corresponding to error 'ER5' does not show any effect in Table 4.4. The error 'ER6' results in the erroneous execution of scenario 1 (i.e. scenario 2 is to be executed in place of scenario 1) and that's why the state transitions are treated as erroneous state transitions.

The rows corresponding to errors 'ER11' (event 'E6') and 'ER12' (event 'E7'), in Table 4.4, do not show any state level effects under any event-sub columns because no component is changing its state during the events 'E6' and 'E7'.

It is to be noted that the 'Event-Errors-Effects-Analysis' for scenario 2 is not shown because there is no state transition occurring in scenario 2 (Figure 3.8 in Chapter 3).

Table 4.4: Event-Errors-Effects-Analysis Scenario 1

Event#	Error#	Effects						
		E1	E2	E3	E4	E5	E6	E7
E1	ER1				X!=x2	Y!=y2		
E2	ER2				X!=x2	Y!=y2		
E2	ER3				X^=x2	Y^=y2		
E3	ER4				X!=x2	Y!=y2		
E3	ER5							
E3	ER6				X^=x2	Y^=y2		
E4	ER7				X!=x2	Y!=y2		
E4	ER8				X!=x2	Y!=y2		
E5	ER9					Y!=y2		
E5	ER10					Y!=y2		
E6	ER11							
E7	ER12							

4.4.5 Time Complexity of the SFMEA Algorithm

(a) Time Complexity of Step I

The algorithmic time complexity of Step I is of the order of ' $O(N1) + O(N2)$ ', where ' $N1$ ' is the number of executable events in a given use-case description file and ' $N2$ ' is the number of 'Event-Sequences' extracted.

(b) Time Complexity of Step II

The running time (i.e. Algorithmic time complexity) of Step II is ' $O(N3 \times N4 \times N5)$ ' where ' $N3$ ' is the number of 'Event-Sequence-State-Transitions', ' $N4$ ' is the number of components for which a state diagrams are drawn and ' $N5$ ' is the average number of executable events in each 'Event-Sequence'.

(c) Time Complexity of Step III

The running time of the Step III is of the order of ' $O(X) + n \times T$ ' where ' X ' is number of executable events in the 'Event-Details' and ' n ' is the number of propagating-types of errors identified manually and ' T ' is the approximate time required first to identify and then to record each such error in the Event-Errors.

(d) Time Complexity of Step IV

The running time of the Step IV is of the order of ' $O(N6 \times N7 \times N8)$ ' where ' $N6$ ' is the number of 'Event-Sequence-State-Transitions' and ' $N7$ ' is the average number of events in each scenario and ' $N8$ ' is the average number of errors for each executable event.

4.5 APPLICATION OF SFMEA ALGORITHM TO SAFETY-CRITICAL SOFTWARE SYSTEMS

The proposed algorithm is applied for two safety-critical applications, namely Insulin Delivery System (IDS) and Rail Track Door Control System (RTCS). The detailed description of these systems and the step-by-step application of the algorithm is given in the following sub-sections.

4.5.1 Motivating Example 1: Insulin Delivery System

The safety-critical insulin delivery system (IDS) case study is selected from the work by Sommerville (Sommerville, 2005). It is an embedded system that is used by diabetes patients to automatically inject the required amount of insulin in the body. The system has a timer which interrupts the system to deliver the required amount of insulin after a fixed time interval. The system has three main components, namely 'Insulin_Controller', 'Sugar_Sensor' and 'Insulin_Pump'. The role of the 'Insulin_Controller' component is to control the operations of the other two components. Whenever instructed by the 'Insulin_Controller', the 'Sugar_Sensor' component measures the current sugar level in the patient's body. The 'Insulin_Pump' component delivers/injects the required amount of insulin in the patient's body.

When an interrupt is received by the IDS the clock timer, the following two tasks are carried-out in sequence.

- (i) The system first measures the current sugar level in the body.

- (ii) If the sugar level is high, then the system computes the amount of insulin to be delivered and instructs the insulin-pump to inject the required amount of insulin in the patient’s body. Otherwise, if the sugar level is low or within acceptable limits, then system displays the same message on the system’s display device.

The use-case description file for this functionality is shown in Figure 4.2.

```

/*****Basic Details*****/
/*      Use Case      :  Deliver Insulin      */
/*      Initiating Actor :  Clock Timer      */
/*      Precondition   :  Sytem is Running    */
/*****/
/*      Event Details      */
clock timer interrupts the system to deliver insulin
The system instructs the sensor to read the current sugar level in blood
IF sugar level in blood is high THEN
    The system computes the amount of insulin dose to be delivered
    System commands the Insulin Pump to deliver the computed amount of insulin
ELSE
    The system displays sugar level ok message on the system display
ENDIF
    
```

Figure 4.2: Use Case Description File for ‘Deliver Insulin’ Use-Case of IDS

Step I: Extracting Event-Sequence for each Scenario

The ‘Event-Details’ extracted from the input file of Figure 4.2 is shown in Table 4.5.

Table 4.5: Event-Details Table for Insulin Delivery System

Event#	Event-Description	Event-Label	Type
E0	clock interrupts the system to deliver insulin	E0	1
E1	system instructs the sensor to read current sugar level in blood	E0,E 1	1
E2	IF sugar level in blood is high THEN	E0,E1,E2	2
E3	system computes the amount of insulin dose to be delivered	E0,E1,E2(T),E3	1
E4	system commands the Insulin Pump to deliver the computed amount of insulin	E0,E 1,E2(T),E3,E4	1
E5	system displays sugar level ok on display	E0,E1,E2(F),E5	1

The ‘Event-Label’ value of event ‘E4’ is ‘E0,E1,E2(T),E3,E4’ and it is not a part of any other event’s ‘Event-Label’ value (Table 4.5). So the events included in this represent a scenario ‘Event-Sequence’ which is {E0,E1,E2,E3,E4}. Similarly, ‘Event-Label’ value of event ‘E5’ is also not part of any other event’s ‘Event-Label’ value. So the another ‘Event-Sequence’ is {E0,E1,E2,E5}. These ‘Event-Sequences’ are shown in Table 4.6 and Table 4.7. The ‘Event-Name’ column is not shown in the respective ‘‘Event-Sequence’ tables.

Table 4.6: Event-Sequence Table for Scenario 1 of Insulin Delivery System

Event#	Precondition	Logical Time
E0		1
E1	E0	2
E2	E0,E1	3
E3	E0,E1,E2(T)	4
E4	E0,E1,E2(T),E3	5

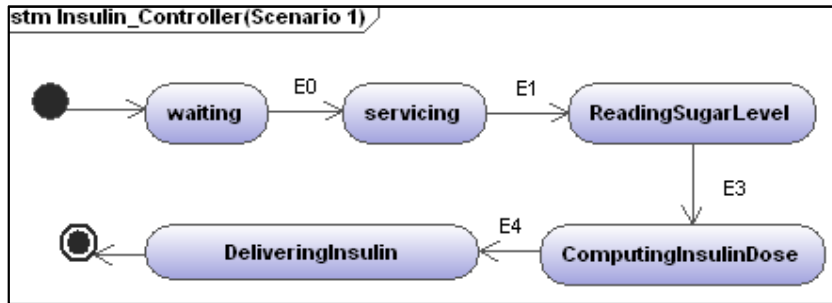
Table 4.7: Event-Sequence Table for Scenario 2 of Insulin Delivery System

Event#	Precondition	Logical Time
E0		1
E1	E0	2
E2	E0,E1	3
E5	E0,E1,E2(F)	4

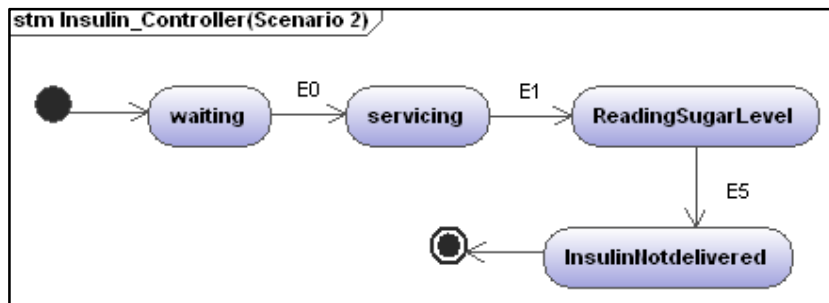
Step II: Identifying Event-Sequence-State-Transitions for each Scenario

The participating components in the IDS system are ‘Insulin-Controller’, ‘Sugar-Sensor’ and ‘Insulin-Pump’. The state diagrams for these components are shown in Figure 4.3. The state transition pattern of the ‘Insulin_Controller’ component is different for two scenarios and that is why two state diagrams are supplied for this component. The state diagram of Figure 4.3(a) for ‘Insulin_Controller’ represents the situation when an insulin is delivered to the patient. Similarly, the state diagram of Figure 4.3(b) for ‘Insulin_Controller’ represents the situation when an insulin is not delivered to the patient. The initial state of the ‘Insulin_Controller’ component is ‘waiting’ whereas the initial states of both the ‘Sugar_Sensor’ and ‘Insulin_Pump’ components are ‘idle’. The

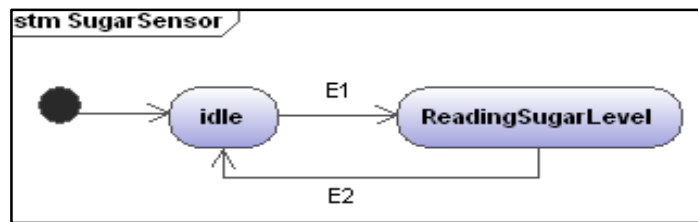
execution of event 'E1' (system instructs the sensor to read the current sugar level in blood, Table 4.5) changes the state of the 'Sugar_Sensor' component from 'idle' to 'ReadingSugarLevel'. Similarly, the execution of event E4 (system commands the Insulin Pump to deliver the computed amount of insulin, Table 4.5) changes the state of the 'Insulin_Pump' component from 'idle' to 'DeliveringInsulin'.



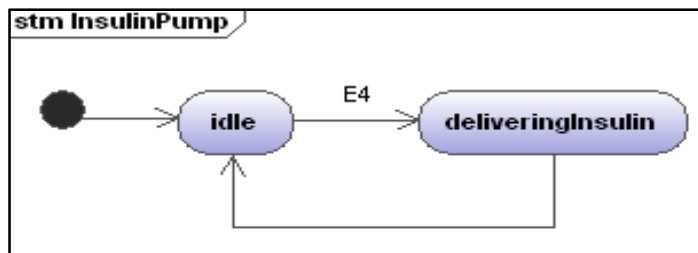
(a) Insulin_Controller State Diagram (When Insulin Delivered)



(b) Insulin_Controller State Diagram (When Insulin Not Delivered)



(c) Sugar_Sensor State Diagram



(d) Insulin_Pump State Diagram

Figure 4.3: State Diagrams for Insulin Delivery System

The application of the second step results in the identification of two ‘Event-Sequence-State-Transitions’ for two scenarios as shown in Table 4.8 and Table 4.9, respectively.

Table 4.8: Event-Sequence-State-Transition Table for Scenario 1

Event#	Insulin-Controller	Sugar-Sensor	Insulin-Pump
E0	servicing	idle	idle
E1	ReadingSugarLevel	ReadingSugarLevel	idle
E2	ReadingSugarLevel	idle	idle
E3	ComputingInsulinDose	idle	idle
E4	DeliveringInsulin	idle	DeliveringInsulin

Table 4.9: Event-Sequence-State-Transition Table for Scenario 2

Event#	Insulin-Controller	Sugar-Sensor	Insulin-Pump
E0	Servicing	idle	idle
E1	ReadingSugarLevel	ReadingSugarLevel	idle
E2	ReadingSugarLevel	idle	idle
E5	InsulinNotDelivered	idle	idle

Step III: Identify Event-Errors for all Scenarios

The application of Step III results in the identification of ‘Event-Errors’ as shown in Table 4.10. The ‘Event-Sequences’ in Table 4.6 and Table 4.7 and UML state diagrams in Figure 4.3 are used as inputs in this step. The following propagating-types of errors are identified manually:

- (i) Error number ER4 at event E1 is a propagating-type error. There is a possibility that because of the fault in the sensor, the sensor reads the wrong current sugar level value.
- (ii) Error number ER9 at event E3. This error can occur because of wrong computation of the value of insulin dose.

Table 4.10: Event-Errors for Insulin Delivery System

Event#	Error#	Type	Error Description
E0	ER1	1	Event E0 fails to execute
E1	ER2	1	Event E1 fails to execute
	ER3	1	sensor failure
	ER4	2	The sensor reads the wrong sugar value
E2	ER5	1	Event E2 fails to execute
	ER6	2	Event E2 is true, but evaluated as false
	ER7	2	Event E2 is false, but evaluated as true
E3	ER8	1	Event E3 fails to execute
	ER9	2	The system computes wrong insulin dose
E4	ER10	1	Event E4 fails to execute
	ER11	1	Insulin pump fails to deliver Insulin
E5	ER12	1	Event E5 fails to execute

Step IV: Performing Event-Errors-Effects-Analysis of each Scenario

The ‘Event-Details’ in Table 4.5, the ‘Event-Sequences’ in Table 4.6 and Table 4.7, the ‘Event-Sequence-State-Transitions’ in Table 4.8 and Table 4.9 and the ‘Event-Errors’ in Table 4.10 are used as inputs to the ‘Event-Errors-Effects-Analysis’ generated for scenario 1 and scenario 2. The results are as shown in Table 4.11 and Table 4.12, respectively. The ‘Effects’ column of Table 4.11 is divided into five event sub-columns labeled as E0, E1, E2, E3, E4 because there are the five events of Scenario 1 (see Table 4.6). Similarly, The ‘Effects’ column of Table 4.12 is divided into four event sub-columns labeled as E0, E1, E2, E5 because these are the four events of Scenario 2 (see Table 4.7).

If more than one component changes their state during the execution of a single event (for example execution of event E1 in Table 4.8 causes a state change in the ‘Insulin-Controller’ and the ‘Sugar-Sensor’ components) then their state level effects in case of a ‘stop-type’ error are joined by an AND operator.

The event E2 (as shown in Table 4.5) is a conditional event and there are two ‘propagating-type’ errors, namely ER6 and ER7 associated with it (as shown in Table 4.10). The error ER6 represents the case when the actual value of the event E2 is true, but it has been evaluated as false. The effects of this error ER6 are observed in the scenario where E2 is false (because the scenario where E2 is true is skipped). That’s why, the row corresponding to event E2 and error ER6 in Table 4.11 does not show any state level effects. On the other hand, the Table 4.12 shows the state level effects of error ER6.

Similarly, ER7 represents the case when the actual value of the event E2 is false, but it has been evaluated as true. The effects of this error ER7 are observed in the scenario where E2 is true. The row corresponding to event E2 and error ER7 in Table 4.11 shows the corresponding state level effects of the error ER7 but does not show any state level effects of the same event and error in Table 4.12.

There is no row corresponding to event E5 in Table 4.11 because event E5 only appears in Scenario 2 and the errors occurring at event E5 can only affect Scenario 2 not Scenario 1. Similarly, there is no row corresponding to event E4 in Table 4.12 because event E4 only appears in Scenario 1 and the errors occurring at event E4 can only affect Scenario 1 not Scenario 2.

Table 4.11: Event-Errors-Effects-Analysis Table for Scenario 1

Event#	Error#	Effects				
		E0	E1	E2	E3	E4
E0	ER1	Insulin-Controller != servicing	Insulin-Controller != ReadingSugarLevel AND Sugar-Sensor != ReadingSugarLevel		Insulin-Controller != ComputingInsulinDose	Insulin-Controller != DeliveringInsulin AND Insulin-Pump!= DeliveringInsulin
E1	ER2		Insulin-Controller != ReadingsugarLevel AND Sugar-Sensor != ReadingSugarLevel		Insulin-Controller != ComputingInsulinDose	Insulin-Controller != DeliveringInsulin AND Insulin-Pump!= DeliveringInsulin
	ER3		Sugar-Sensor != ReadingSugarLevel		Insulin- Controller != ComputingInsulinDose	Insulin-Controller != DeliveringInsulin AND Insulin-Pump!= DeliveringInsulin
	ER4				Insulin-Controller ^= ComputingInsulinDose	Insulin-Controller ^= DeliveringInsulin AND Insulin-Pump^= DeliveringInsulin
E2	ER5				Insulin-Controller != ComputingInsulinDose	Insulin-Controller != DeliveringInsulin AND Insulin-Pump!= DeliveringInsulin

Event#	Error#	Effects				
		E0	E1	E2	E3	E4
	ER6					
	ER7				Insulin-Controller ^= ComputingInsulinDose	Insulin-Controller ^= DeliveringInsulin AND Insulin-Pump ^= DeliveringInsulin
E3	ER8				Insulin-Controller != ComputingInsulinDose	Insulin-Controller != DeliveringInsulin AND Insulin-Pump != DeliveringInsulin
	ER9				Insuline-Controller ^= ComputingInsulinDose	Insulin-Controller ^= DeliveringInsulin AND Insulin-Pump ^= DeliveringInsulin
E4	ER10					Insulin-Controller != DeliveringInsulin AND Insulin-Pump != DeliveringInsulin
	ER11					Insulin-Pump != DeliveringInsulin

Table 4.12: Event-Errors-Effects-Analysis Table for Scenario 2

Event#	Error#	Effects			
		E0	E1	E2	E5
E0	ER1	Insulin-Controller!= servicing	Insulin-Controller != ReadingSugarLevel AND Sugar-Sensor != ReadingSugarLevel		Insulin-Controller!= InsulinNotdelivered
E1	ER2		Insulin-Controller != ReadingSugarLevel AND Sugar-Sensor != ReadingSugarLevel		Insulin-Controller!= InsulinNotdelivered
	ER3		Sugar-Sensor != ReadingSugarLevel		Insulin-Controller!= InsulinNotdelivered
	ER4				Insulin-Controller^= InsulinNotdelivered
E2	ER5				Insulin-Controller!= InsulinNotdelivered
	ER6				Insulin-Controller^= InsulinNotdelivered
	ER7				
E5	E12				Insulin-Controller!= InsulinNotdelivered

4.5.2 Motivating Example 2: Railway Track Door Control System (RTCS)

This example is described in Chapter 3 for SFTA application. The use-case description of this case study is given in Section 3.8 in Chapter 3. There are two participating components namely ‘Track_Door’ and ‘Track_Signal’ are considered in the current approach. The state diagrams for these two components are shown in Figure 3.21 in Chapter 3.

The ‘Event-Sequences’ of this application are shown in Table 3.22 and Table 3.23 and the ‘Event-Sequence-State-Transitions’ for two scenarios are shown in Table 3.24 and Table 3.25 in Chapter3.

The ‘Event-Errors’ identified for this application are shown in Table 4.13.

Table 4.13: Event-Errors Identified For RTCS Application

Event#	Error#	Type	Error Description
E1	ER1	1	Event E1 fails to execute
E2	ER2	1	Event E2 fails to execute
	ER3	1	Error in Track Door Component
E3	ER4	1	Event E3 fails to execute
	ER5	2	Event E3 is False but evaluated as True
	ER6	2	Event E3 is true but evaluated as False
E4	ER7	1	Event E4 fails to execute
	ER8	1	Error in Track_signal Component
E5	ER9	1	Event E5 fails to execute
E6	ER10	1	Event E6 fails to execute

The ‘Event-Errors-Effects-Analysis’ of two scenarios are shown in Table 4.14 and Table 4.15.

Table 4.14: Event-Errors-Effects-Analysis For Scenario 1

Event#	Error#	Effects				
		E1	E2	E3	E4	E5
E1	ER1		Track_Door!=closed		Track_Signal != green	
E2	ER2		Track_Door!=closed		Track_Signal != green	
	ER3		Track_Door!=closed		Track_Signal != green	
E3	ER4				Track_Signal != green	
	ER5				Track_Signal ^= green	
	ER6					
E4	ER7				Track_Signal != green	
	ER8				Track_Signal != green	
E5	ER9					

Table 4.15: Event-Errors-Effects-Analysis For Scenario 2

Event#	Error#	Effects			
		E1	E2	E3	E6
E1	ER1		Track_Door!=closed		
E2	ER2		Track_Door!=closed		
	ER3		Track_Door!=closed		
E3	ER4				
	ER5				
	ER6				
E6	ER10				

4.5.3 Analysis of Results

The results of the SFMEA algorithm help the analyst in forecasting beforehand the erroneous state level effects caused by various event-related errors. A stop type of event-related error prevents the components from changing their expected state transitions whereas a propagating type of event-related error erroneously changes the states of the components. There exists many-to-many mapping between event-related errors and erroneous state level effects. A single event-related error can cause multiple state level effects and in the same way, a single erroneous state level effect can be caused by multiple errors. Any type of event-related error occurring at the earlier events causes more erroneous state level effects than the event-related error occurring at the later events. For example, in Table 4.11, the event-related error 'ER1' at event 'E0' causes state level effects under the event sub-columns 'E0', 'E1', 'E3' and 'E4'. But, the event-related error 'ER10' at event 'E4' causes state level effect only at event 'E4'. Similar is the case for errors 'ER1' and 'ER7' in Table 4.11.

Consider the Table 4.14. The erroneous state level effect 'Track_Door! =closed' is caused by three event-related errors, namely ER1, ER2 and ER3. The erroneous state level effect 'Track_Signal!= green', in Table 4.14, is caused by six event-related errors namely ER1, ER2, ER3, ER4, ER7 and ER8. Similar cases can be found in Table 4.11 and Table 4.12 also.

The results of the 'Event-Errors-Effects-Analysis' can be used by the analyst to determine the overall mapping between the event-related errors and their erroneous state level effects. For example, consider the Table 4.14 and Table 4.12. There exist three distinct erroneous state level effects (as 'Track_Door! =closed', 'Track_Signal != green' and 'Track_Signal ^= green') which are caused by nine event-related errors (ER1,ER2,...,ER9). The mapping between these effects and event-related errors are shown in Figure 4.4. When all the erroneous state level effects are known, the analyst assigns a severity rating to each effect manually. The knowledge and domain expertise of the analysts plays an important role in it. National Aeronautics Space Administration (NASA) recommends four types of severity ratings as Catastrophic, Critical, Moderate and Negligible (NASA-GB-8719.13, 2004). The event-related errors responsible for 'Catastrophic' and 'Critical' effects are considered more serious than the event-related errors responsible for 'Moderate' and 'Negligible' effects.

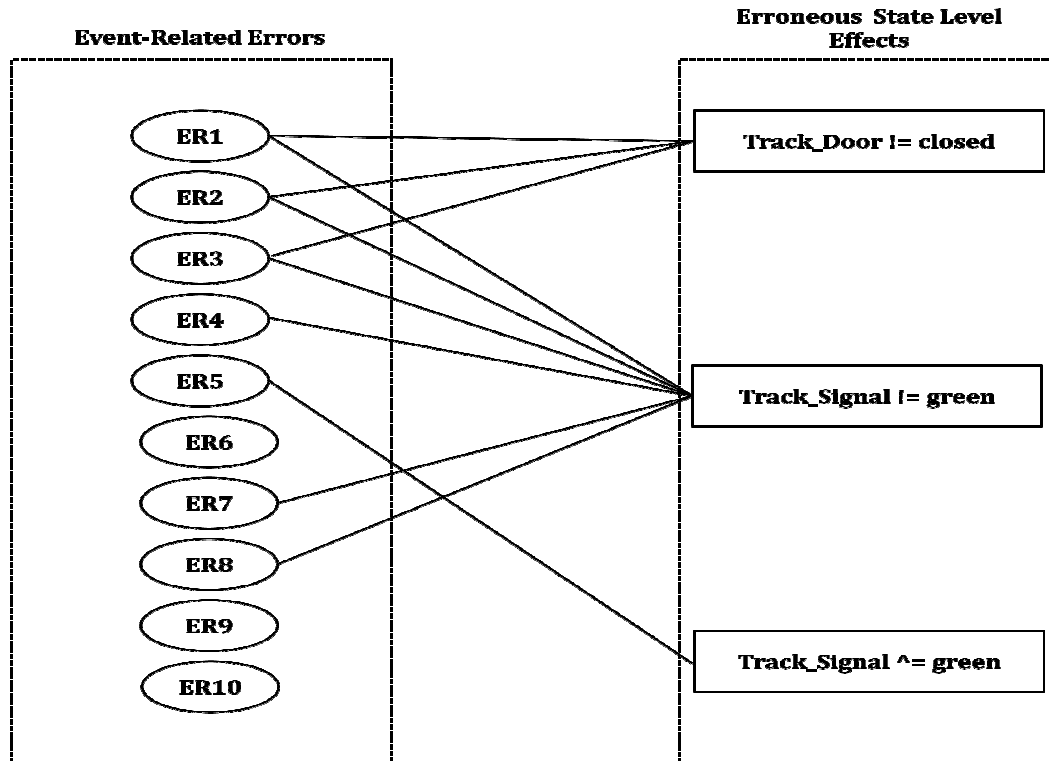


Figure 4.4: Mapping of Event-Related Errors and Erroneous State Level Effects of RTCS

4.6 COMPARISON OF SFTA AND SFMEA APPROACHES

Gupta (Gupta et al, 2012) introduced an eight step integrated SFMEA and SFTA approach by taking the formal description of the given use-case functionality as an input. However, the application of the approach is manual. Similarly, the work reported by Tiwari (Tiwari et al, 2012) also described an integrated SFTA and SFMEA application approach that takes formal use-case description as the input.

The application of the presented SFMEA approach actually augments the results of the SFTA approach discussed in Chapter 3 in two ways.

- (i) The presented SFMEA approach assists in the completeness of the fault trees drawn using the SFTA approach of Chapter 3 as follows

For example, consider the results of the application of the presented SFMEA approach on Rail Track Door Control System (RTCS) applications as shown in Table 4.14 and the fault tree for the hazardous state 'Track_Door!=closed AND Track_Signal=green' as shown in Figure 3.23. The fault tree of Figure 3.23 has only two errors responsible for error state 'Track_Door!=closed'. But as per the SFMEA analysis results of Table 4.14,

the same error state can occur because of three errors namely 'ER1', 'ER2' and 'ER3'. Recall that in the fault tree for the hazardous-state 'Track_Door !=closed AND Track_Signal' as shown in Figure 3.23, only two state-related errors are considered for the erroneous state 'Track_Door !=closed' and these are 'ER1' and 'ER2'. However, the application of the SFMEA approach has identified an extra error in the form of 'ER3' also. Therefore, the application of the SFMEA approach actually helps in the completeness of fault trees constructed using SFTA approach.

- (ii) The presented SFMEA approach is used to construct fault trees for computational types of hazardous-states also which is not possible by the application of the SFTA approach of Chapter 3. An example in support of this is given below.

For example, consider the hazardous-state for the Insulin Delivery System (IDS) where a wrong amount of insulin is delivered in the patient's body. The construction of the fault tree for this hazardous-state is not possible by the SFTA approach of Chapter 3. However, the fault tree for hazardous-state can be constructed using the results of the presented SFMEA approach as follows.

The effect entry 'InsulinPump^= DeliveringInsulin' in Table 4.11(row of error 'ER4') indicates that the state of the insulin pump component is erroneously changed to 'DeliveringInsulin'. It actually represents a situation where an insulin is delivered in an erroneous fashion. There are three errors namely 'ER4', 'ER7' and 'ER9' that cause this type of effect. So the fault tree for the hazardous-state 'InsulinPump^= DeliveringInsulin' contains three errors that are joined via an 'OR' gate.

The main strength(s) of the approach are as follows:

- (i) The approach investigates the erroneous effects of every event-related error. Recall that in the SFTA approach of Section 3.1, the event-related of only state transition events are considered.
- (ii) The approach augments the application of the SFTA approach presented and discussed in Section 3.1.
- (iii) The proposed SFMEA approach is semi-automatic and the whole error analysis process takes a considerable less amount of time as compared to other available non-automatic methods (Wentao and Hong 2009, Gupta et al 2012, Tiwari et al 2012).

The main weaknesses of the approach are as follows:

- (i) It investigates the state level effects of only one event-related error at a time.
- (ii) The propagating-type of errors for normal events are to be identified manually and because of that, the application process is semi-automatic.
- (iii) The approach in the present form does not handle concurrent state transitions of the participating components.

Software Fault Tree Analysis Approach for Object-Oriented Design Phase

It was noted in Chapter 2 that the existing analysis tools for object-oriented design (OOD) phase do not provide the support for either the automated or semi-automated application of the SFTA approach. This chapter presents a SFTA approach for the hazard analysis of the object-oriented design models. In the proposed approach, the events corresponding to the messages of a given sequence diagram are mapped against the states of the participating objects and using these the software fault tree (SFT) for the selected hazardous-state of the OOD model is constructed. The proposed SFTA approach is semi-automated and the algorithm is divided into four steps where the software fault tree construction step is automated. The UML sequence diagram for a selected use-case scenario and the UML state diagrams of the participating objects are required as the inputs. It requires proper tagging of both the sequence and state diagrams. The approach has been validated by applying it on the UML design models of two use-cases, namely 'Request Elevator' and 'Stop Elevator' of an Elevator Control System (ECS) application, used earlier in Chapter 3.

5.1 OBJECT-ORIENTED DESIGN PROCESS

Object-oriented design is a common approach to software design where a particular software problem is divided into a system of collaborating/interacting objects. The use-case models are developed during the object-oriented requirements analysis phase. The use-case realization template written for each use-case functionality is translated into a sequence diagram. The outputs of the object-oriented design phase include the following:

- (i) A set of classes, their attributes and the responsibilities/operations.
- (ii) A sequence diagrams for each use-case functionality. Each sequence diagram gives the information about the objects, which are collaborating with each other in order to realize the selected use-case functionality.
- (iii) A set of state diagrams, where each state diagram depicts the state transition pattern of a single object. It is to be noted that a state diagram focuses on the state transition pattern of a single object, whereas the sequence diagram focuses on all the objects that are required to realize the selected use-case functionality.

5.2 OBJECTIVE OF THE PROPOSED SFTA APPROACH

The focus of SFTA research efforts in the object-oriented software design phase is either the automatic or semi-automatic construction of a software fault tree for a selected hazardous-state of the system directly from the given object-oriented design models. However, the efforts have not reported as success so far.

The work of Pai and Dugan (Pai and Dugan, 2002) presents an approach to automatically construct the dynamic fault trees (DFTs) from UML class, activity and deployment diagram(s). However, the models used by Pai and Dugan do not represent any functional aspect of the system. Rather, UML has been used to model certain fault tolerant features of hardware systems such as redundancy and error-propagation. The objective of their work was reliability assessment and not the hazard analysis. Similarly, an approach to synthesize fault tree(s) for reliability analysis from architectural model has been proposed by Lauer and German (Lauer and German, 2011). The work described by Massood (Massood et al., 2002, 2003) provided a partial paradigm in the form of guidelines for mapping sequence, state and activity charts to corresponding fault trees, but the application process was manual as the possible message type errors have to be identified manually. The application of the SFTA approach on the UML sequence and state diagrams is manual, error-prone and time-consuming. The proposed algorithm integrates and semi-automates the application of the SFTA approach in object-oriented software design phase and overcomes all these limitations.

5.3 ASSUMPTIONS FOR THE PROPOSED APPROACH

The assumptions made for the proposed approach are explained below.

- (i) ***The sequence and state diagrams are complete and correct and are drawn for a single scenario***

The algorithm is based on the assumption that the supplied sequence and state diagrams are correct and complete and represents the functionality of a single scenario of a given use-case functionality. The completeness of the sequence diagram means that '*no message and class has been missed-out*'. The correctness of the sequence diagram means '*the sequence in which the various messages are exchanged among the collaborating/interacting objects, is specified correctly*'. It is also assumed that the UML state diagrams are correct (*state transition events are correctly specified*) and complete

(no state transition for any object is missed-out). The correctness and completeness of the inputs are required for the correctness and completeness of the constructed software fault trees. The presence of a wrong message in the sequence diagram or a wrong state transition in the state diagram will result in erroneous SFT.

(ii) No participating object experience concurrent state transitions

It is assumed that *corresponding to any state transition event there is a single state transition experienced by any participating object*. This assumption is same as in Chapter 3.

(iii) The information about the type of object participating in the interaction is embedded in the object name.

The proposed SFTA algorithm assumes that the information about three categories of objects, namely Controller, Device and Interface types, is embedded in the name of the object itself. The controller type of objects control the functionalities of device type objects. The interface type objects act as a communication channel to transfer the messages between controller and device types of objects. The algorithm uses three special tags, namely ((Controller)),((Device)),((Interface)) along with object name to represent the controller, device and interface types of objects. For example, Elevator((Controller)) is a controller type object, Door((Device)) is a device type object and Motor((Interface)) is an interface type object.

(iv) Hazardous-state is expressed using states of only two device types of objects only

It is also assumed that the hazardous-state, for which a software fault tree (SFT) is to be constructed, is expressed using the states of two device type objects only.

5.4 OVERVIEW OF THE PROPOSED SFTA ALGORITHM

The proposed SFTA algorithm is divided into four steps to construct the software fault tree (SFT) from the scenario sequence and its associated state diagrams.

The first step extracts the attributes of each message from the scenario sequence diagram. The attributes that are extracted and computed for each message are shown in Table 5.1. The algorithm takes the scenario sequence diagram and associated state diagrams as inputs in machine readable format i.e. XMI (XML Metadata Interchange) format. The Altova UML(Altova-UModel, 2014) tool is used to draw the required

sequence and state diagrams and each of them is exported to XMI format using the same tool.

Table 5.1: Attributes Extracted From Messages

Message#	Name	Precondition	Type	From	To	Send/Receive Event Pair
<<Unique Message Number>>	<<Name of the Message as used in the sequence diagram>>	<<Precondition of the Message as used in the sequence diagram>>	<<Type of the Message is 1 for Asynchronous Send, 2 for Synchronous Send, 3 For Asynchronous Reply, 4 for Synchronous Reply >>	<<Name of the Sender Object>>	<<Name of the Receiver Object>>	<<Send event and Receive event pairs>>

The second step takes the ‘Message-Sequence’ extracted in the first step and the UML state diagram XMI files as the inputs and identifies the ‘Event-Sequence-State-Transitions’ of a scenario. The structure of the ‘Event-Sequence-State-Transition’ depends upon the number of objects for which the state diagrams are supplied as inputs. If the state diagrams for two arbitrary objects, namely ‘X’ and ‘Y’ are supplied as inputs in this step, then the structure of the ‘Event-Sequence-State-Transition’ is as shown in Table 5.2.

Table 5.2: Structure of ‘Event-Sequence-State-Transition’ Table

Event	Type	Timestamp	X	Y
<<Event of a message>>	<<Type of event >>	<<Time of occurrence of the event>>	<<State of X during the of event>>	<<State of Y during theevent>>

In the third step, for a given ‘hazardous-state’ of the system, the ‘Message-Sequence’ and the ‘Event-Sequence-State-Transitions’ are analyzed to generate the software fault tree (SFT) as an XML file. The actual SFT is drawn from this file by the fault tree creation tool named FaultCAT (FaultCAT,2003) in the next step.

An overview of the steps of the SFTA approach is shown in Figure 5.1 and the SFTA algorithm is explained in the next section.

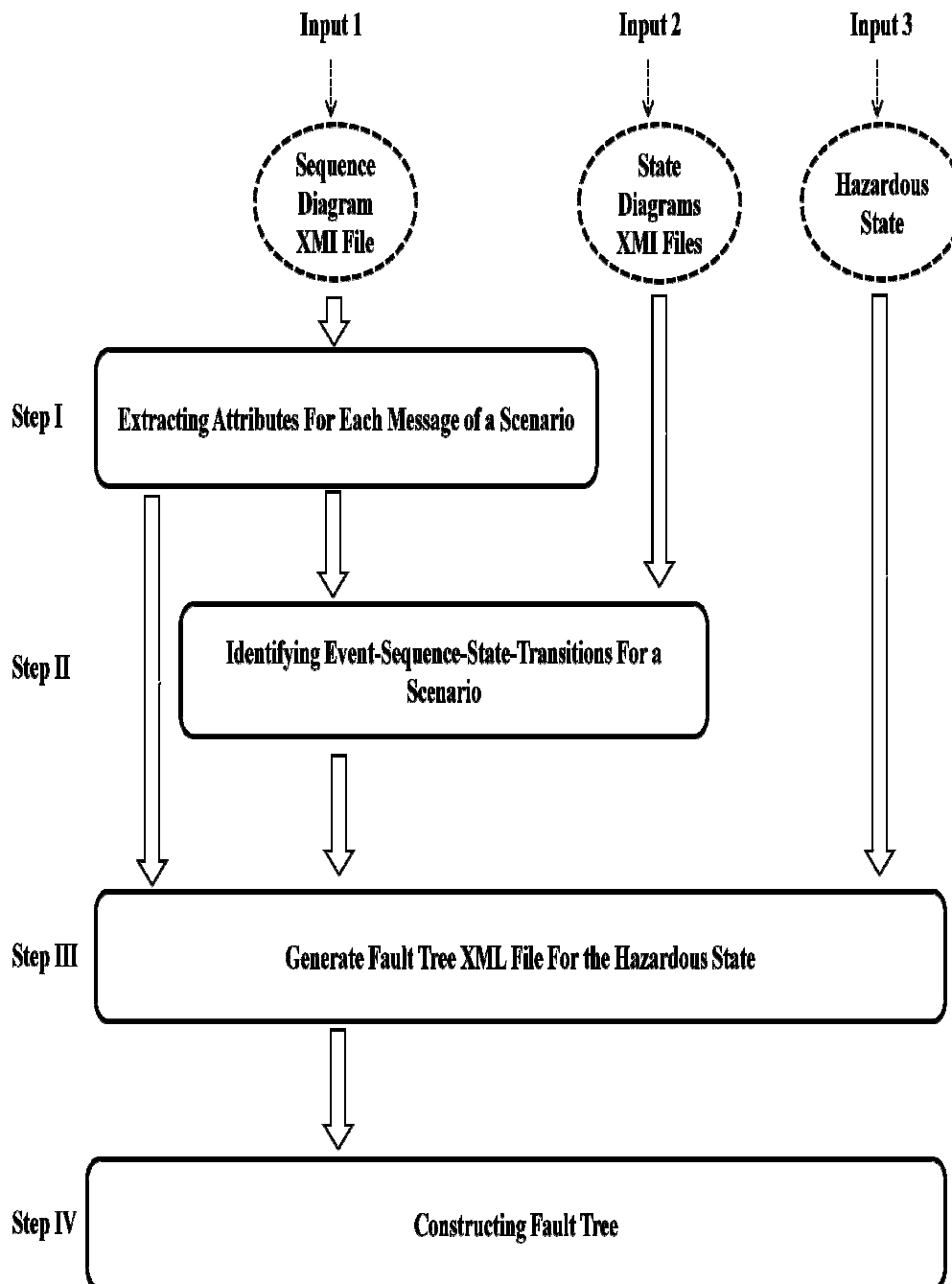


Figure 5.1: Overview of the Proposed SFTA Algorithm for Object-Oriented Design Phase

5.5 THE PROPOSED SFTA ALGORITHM

5.5.1 Step I: Extracting Attributes of each Message from a Scenario Sequence Diagram

This step extracts the information for each message from the sequence diagram XMI file. The extracted information is stored in tabular form as shown in Table 5.1. Each message is associated with two types of events, namely send-event and receive-event. The send event occurs on the lifeline of the sender object, whereas the receive event occurs on the lifeline of the receiver object. So, for each type of message (send type or reply type), a unique event pair named ‘Send/Receive Event Pair’ is generated in this step.

The pseudo code of this step is given below.

Procedure *populateMessageSequence()*

Input *Sequence Diagrams XMI File*

1. *Create Message-Sequence Table with Structure as shown in Table 5.1*
2. **FOR** *each message tag in the Sequence Diagrams XMI File*
 - (i) *Extract the attributes for the message*
 - (ii) *Generate a unique ‘Send/Receive Event Pair’ for the message*

ENDFOR

5.5.2 Step II: Identifying ‘Event-Sequence-State-Transitions’ for a Scenario

This step identifies ‘Event-Sequence-State-Transition’ of the scenario by using the ‘Message-Sequence’ (of Step I) and the state diagram XMI files of the participating objects as inputs. The structure of the ‘Event-Sequence-State-Transition’ table is divided in two parts, namely ‘Event-Sequence’ part and ‘State-Transition’ part. The ‘Event-Sequence’ part has three fields/columns, namely ‘Event’ (name of the event), ‘Type’ (type of the event) and ‘Logical Time’ (time when the event occurred) and this part is identified from the information given in ‘Message-Sequence’ of Step I. The ‘State-Transition’ part is identified from the information given in the state diagram XMI files.

The ‘Message-Sequence’ and ‘State-Transition’ parts are identified in two sub-steps as follows.

Step II(a) Identifying ‘Event-Sequence’ of a Scenario

The ‘Message-Sequence’ table of Step I is sorted on a unique sequence number value assigned to every message and for every message a unique send/receive event pair has been created. This step first appends the values for each send/receive event pair for every message in the ‘Event’ column of the ‘Event-Sequence-State-Transition’. After that the values for ‘Type’ and ‘Logical Time’ fields are assigned. The ‘Type’ field value of any event is either 1 (if the event is a send event of sending type message) or 2 (if the event is a receive event of a send type message) or 3 (if the event is a send event of a reply type message) or 4 (if the event is a receive event of a reply type message).

Every event is assigned a unique integer value known as its logical time (initialized to 0 at the start of this step) and it is incremented by 1 between the occurrences of two successive events.

The pseudo code for this step is given below

Procedure *populateEventSequence()*

Input *Event-Sequence table and State Diagrams XMI (seq.xmi) Files*

logical_time = 0;

FOR each message ‘msg’ of the Message-Sequence table

- (i) Read the send event ‘es’ for the message ‘msg’ and append it in the Event-Sequence-State-Transition table;
- (ii) Assign type for ‘es’ as per above mentioned description;
- (iii) Assign the logical time value for ‘es’ as *logical_time+1*; increment *logical_time* by 1;
- (iv) Repeat the above steps (1)–(3) for receive event (er)

ENDFOR

Step II(b) Identifying ‘State-Transitions’ of a Scenario

This sub-step identifies the ‘State-Transition’ part of the ‘Event-Sequence-State-Transition’ from the information given in the state diagram XMI files. The state transition of any participating object takes place either during the send event or during the receive event of a message as follows:

Suppose, an object ‘O’ is making a state transition to a new state ‘S’ upon the occurrence of the message ‘M’. If object ‘O’ is not the sender of the message ‘M’ in the given sequence diagram, then object ‘O’ will make a transition to state ‘S’ at the occurrence of the receive event of message ‘M’ otherwise (if object ‘O’ is the sender of the message

'M') object 'O' will make a transition to state 'S' at the occurrence of the send event of message 'M'.

The pseudo code for this step is given below.

```

Procedure    populateState-Transition()
FOR each state chart XMI file of step 2
    read current_object; /* name of the object represented by state chart XMI file */
    read initial_state of current_object; /* initial state of the object */
    FOR each message 'msg' of Message-Sequence table
        IF message 'msg' is responsible for any state transition in the XMI file THEN
            read next_state of the current_object;
            IF current_object is not the sender of message 'msg' THEN
                update the state column of current_object in Event-
                Sequence-State-Transition table with next_state
                corresponding to receive event of message 'msg';
            ELSE
                update the state column of current_object in Event-
                Sequence-State-Transition table with next_state
                corresponding to send event of message 'msg';
            ENDIF
        ENDIF
    ENDFOR
ENDFOR
FOR event of Event-Sequence-State-Transition table
    IF current_object state column is empty THEN
        update the current_object column with initial_state;
    ELSE
        read the state stored under into current_object column initial_state
    ENDIF
ENDFOR

```

5.5.3 Step III: Generating Fault Tree XML File for Selected Hazardous-State

This step generates an XML file which represents the software fault tree for the selected hazardous-state. The hazardous-state of the system, the 'Message-Sequence' extracted in Step I and the 'Event-Sequence-State-Transition' identified in Step II are used as input(s) in this step.

As mentioned earlier, the proposed approach constructs the software fault tree for the hazardous-state involving the states of two device type objects only. Assume 'a' and 'b'

are the valid states of two device type objects, namely A((Device)) and B((Device)) respectively. There can be four possible ways of expressing the hazardous-state of the system.

Case 1 A((Device)) != a AND B((Device)) = b

Suppose at time ‘T1’ the state of the A((Device)) object is changed to ‘a’ and at time ‘T2’ the state of the B((Device)) object is changed to ‘b’ and ‘T1’ is less than ‘T2’. This case represents a hazardous situation where the required state transition of the A((Device)) object to state ‘a’ (which is supposed to occur earlier) has not occurred at all, whereas the state transition of B((Device)) object to state ‘b’ (which is supposed to occur later) has occurred. The ‘AND’ operator is used here to join the states from various objects.

The pseudo code of the fault tree construction for Case 1 hazardous-state is given below:

Procedure createFaultTree()

Input(s) Message-Sequence Table, Event-Sequence-State-Transition Table and Hazardous-State

Output faulttree.xml File

(a) Create Fault Tree For A((Device)) != a

- 1 *Identify messages responsible for A((device)) = a from message-sequence table using event-sequence-state-transition table as input.*
- 2 *Create a basic error event named !(M) for each such message where M is the message number of the message. /* the error event !(M) indicates that the message M has not been sent whereas it is to be sent */.*
- 3 *If a message has a mentioned precondition, then create a basic error event named ^(preC) where preC is the precondition of the message. /* error event ^(preC) represents a situation where precondition preC has been wrongly evaluated as false whereas it is true */.*
- 4 *If number of basic error events created in steps (2) and (3) above are more than 1 then join all the error events by an AND gate and feed the output of this AND gate to an intermediate error event named A((device)) != a else use wire gate and feed the output of this gate to an intermediate event named A((device)) != a.*

(b) Create Fault Tree For B((Device)) = b

- 1 *Identify messages responsible for B((device)) = b from message-sequence table using event-sequence-state-transition table as input.*

- 2 *Create a basic error event named (M) for each such message where M is the message number of the message. /* the error event (M) indicates that the message M has been wrongly sent, whereas it is not to be sent */.*
 - 3 *If a message has a mentioned precondition, then create a basic error event named (preC) where preC is the precondition of the message. /* error event (preC) represents a situation where precondition preC has been wrongly evaluated as true whereas it is false */.*
 - 4 *If the number of basic error events created in steps (2) and (3) above are more than 1 then join all the error events by an AND gate and feed the output of this gate to an intermediate event named $B((device)) = b$ else use wire gate and feed the output of this gate to an intermediate event named $B(device) != b$.*
- (c) Join the fault trees created in steps (a) and (b) above by an AND gate and feed the output of this gate to a hazardous state named $A((device)) != a$ and $B((device)) = b$.

Case 2 $A((Device)) = a$ AND $B((Device)) != b$

The fault tree construction process for this type of hazardous-state is similar to Case 1 hazardous-state, but in this case the state transition of $A((Device))$ object to state 'a' has occurred, but the required state transition of $B((Device))$ object to state 'b' has not occurred.

The pseudo code of the fault tree construction for Case 2 hazardous-state is given below:

- (a) *Construct fault tree for $A((Device)) = a$
Use the fault tree construction steps of $B((Device)) = b$ of Case 1.*
- (b) *Construct fault tree for $B((Device)) != b$
Use the fault tree construction steps of $A((Device)) != a$ of Case 1.*
- (c) *Join the fault trees created in steps (a) and (b) above by an AND gate and feed the output of this gate to a hazardous state named $A((Device)) = a$ and $B((Device)) != b$*

Case 3 $A((Device)) != a$ AND $B((Device)) != b$

This case is similar to Case 1, but in this case, both the required state transitions of $A((device))$ and $B((device))$ objects to states 'a' and 'b' respectively have not occurred.

The pseudo code of the fault tree construction for Case 3 hazardous-state is given below:

- (a) *Construct fault tree for $A((Device)) != a$ using the fault tree construction steps of $A((Device)) != a$ of Case 1*
- (b) *Construct fault tree for $B((Device)) != b$ using the fault tree construction steps of $A((Device)) != a$ of Case 1*
- (c) *Join the fault trees created in steps (a) and (b) above by an AND gate and feed the output of this gate to a hazardous state named $A((Device)) != a$ and $B((Device)) != b$.*

Case 4 $A((Device)) = a$ AND $B((Device)) = b$

This case is different from the previous three cases. According to this case, for the correct operation of the system, at any point of time the system should not have both $A((Device))$ and $B((Device))$ objects in states 'a' and 'b' respectively. As long as the object $A((Device))$ is in the state 'a', the $B((Device))$ object should not be allowed to change its state to 'b' state and vice versa.

The pseudo code of the fault tree construction for Case 4 hazardous-state is given below:

(a) *Select time 't1' such that $A((Device)) = a$ for the first time*

(b) *Select time 't2' such that $B((Device)) = b$ for the first time*

(c) **IF** $t1$ is less than $t2$ **THEN**

Select the first time $t3$ ($t1 < t3 < t2$) such that $A((Device)) \neq a$;

Read the state (say 'x') of $A((Device))$ at time $t3$;

Use Case 1 procedure for the hazardous state $A((Device)) \neq x$ and $B((Device)) = b$;

ELSE

Select the first time $t3$ ($t2 < t3 < t1$) such that $B((Device)) \neq b$;

Read the state (say 'x') of $B((Device))$ at time $t3$;

Use Case 1 procedure for the hazardous state $B((Device)) \neq x$ and $A((Device)) = a$;

ENDIF

5.5.4 Step IV: Constructing Fault Tree

In this step, the software fault tree is drawn in graphical form by using the fault tree XML file created in Step III as an input to the FaultCAT tool.

5.5.5 Time Complexity of the SFTA Algorithm**(a) Time Complexity of Step I**

The running time, i.e. the algorithmic time complexity of Step I is of the order of ' $O(N1)$ ', where ' $N1$ ' is the number of messages in the sequence diagram.

(b) Time Complexity of Step II

The running time, i.e. the algorithmic time complexity of Step II(a) is of the order of $O(N1)$, where ' $N1$ ' is the number of messages in the 'Message-Sequence' table.

The running time, i.e. the algorithmic time complexity of Step II(b) is of the order of ' $O(N1 \times N2)$ ', where ' $N1$ ' is the number of messages in the 'Message-Sequence' table and ' $N2$ ' is the number of components for which the state diagrams are drawn in this step.

(c) Time Complexity of Step III

The running time, i.e. the algorithmic time complexity of Step III is of the order of $O(N^3)$, where 'N3' is the number of events in the 'Event-Sequence-State-Transitions' of a scenario.

5.5.6 Formatting of Inputs

In the SFTA algorithm described above, it is assumed that the three inputs required in the algorithm requires are supplied in some specific representations as explained in the following sub-sections.

(a) Sequence Diagram Representation

The sequence diagram of the selected functionality should satisfy the following:

- (i) The name of any send-type message should have the following format:

$$[preC] \{M\} name-of-message(parameter-list)$$

where '*preC*' is the precondition that must be true before the message is sent and '*M*' is a unique message number assigned to that message. Message number '*M*' has a structure of the form '*Ad*' where '*A*' is any capital alphabet and '*d*' is a unique message sequence number. The Altova UModel tool provides a feature to automatically assign a unique integer sequence number to every message. This sequence number assigned by the tool should be used for assigning a unique message number to each message of the sequence diagram by inserting this sequence number after any capital alphabet letter and this alphabet letter should be same across the whole sequence diagram e.g., A1, B9 etc.

- (ii) Similarly, the name of the reply-type message (either synchronous or asynchronous) should have the format: $\{M\} name-of-message$, where M is a unique message number assigned to the reply message.
- (iii) The sequence diagram should be drawn without using an '*alt*' block feature as shown in sequence diagram of Figure 5.2. In the Figure 5.2, the message M1 is sent only if the condition $a = 0$ is true, otherwise the message M2 will be sent. The presence of each '*alt*' block indicates the presence of an alternate flow of actions of the selected functionality. But, while drawing a sequence diagram for any use-case functionality, only one of the two possible paths should be selected so that it represents the functionality of a single scenario, i.e., sequence diagram should be drawn either for true condition scenario or for false condition scenario.
- (iv) The object type information is to be included in the object name itself by appending special tag(s) such as $((Controller))$, $((Interface))$ and $((Device))$ after the object

name. For example, *Elevator((Controller))* indicates a controller type object, *Door((Device))* indicates device type object and *Motor((Interface))* indicates an interface type object. The controller type objects are responsible for controlling the state transitions of both the device type objects and the interface type objects. The interface type objects are used as a message exchange medium between controller and device type objects.

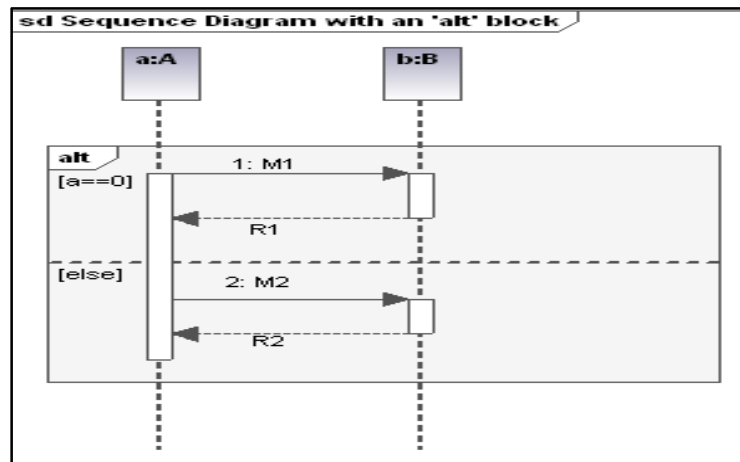


Figure 5.2: A Sample Sequence Diagram using 'alt' block

(b) State Diagram Representation

The state diagrams drawn for the participating objects should satisfy the following:

- (i) The state diagrams drawn for the selected collaborating objects should use the unique message numbers assigned to various messages as state transition events.
- (ii) For every state of a device type object, there are to be defined two states for the controller type object and the names of these states should be as per the guidelines mentioned below.

Consider a controller type object named *C((Controller))* and device type object named *D((Device))*. Suppose *C((Controller))* object issues a command to *D((Device))* object to change its state to a new state named 's'. The *D((Device))* object will make a state transition to state 's' after receiving the command. After successful state transition, the *D((Device))* object will send the reply back to *C((Controller))* object. In this case, there are to be defined two states for *C ((Controller))* object. The name of the first state of the *C((Controller))* object is to be named as 'prepare(*D((Device)))* = s' and it should occur when *C((Controller))* object issues the command. The name of the second state transition is to be named as '*D((Device))* = s' and it should occur when *C((Controller))* object receives the reply message back from *D((Device))* object.

(c) Hazardous-State Representation

The hazardous-state for which a software fault tree is to be constructed is to be expressed in terms of the states of two device types objects only. The hazardous-state involving the states of device type objects can have four forms as discussed in Section 5.5.3.

5.6 APPLICATION OF THE ALGORITHM IN SAFETY-CRITICAL APPLICATION: ELEVATOR CONTROL SYSTEM

The proposed SFTA application approach is applied on two use-cases, namely *dispatch elevator* and *stop elevator* and these scenarios are part of the design of an elevator control system application described by Gomaa (Gomaa, 2000).

5.6.1 Dispatch Elevator Scenario

The ‘Dispatch Elevator’ use-case functionality is required in the ECS application to dispatch an elevator to a particular floor in response to a user request. The sequence diagram drawn for the ‘Dispatch Elevator’ use-case is shown in Figure 5.3. The objects that are participating in this scenario are: ‘:ElevatorStatusPlan’, ‘:Elevator((Controller))’, ‘:Motor((Interface))’, ‘:Motor((Device))’, ‘:Door((Interface))’ and ‘:Door((Device))’. The format used to represent the name of the object is ‘*object-name:class-name*’, where object-name is the name of the object and is optional and ‘class-name’ represents the name of the class to which the object belongs.

The information about the floors where an elevator has to visit is maintained by the ‘ElevatorStatusPlan’ object. The ‘ElevatorStatusPlan’ object instructs the ‘Elevator((Controller))’ object to move the elevator to a particular floor number. The ‘:Elevator((Controller))’ object is used to control the operations of the ‘:Door((Device))’ and the ‘:Motor((Device))’ objects. The ‘:Motor((Interface))’ object is used as an interface to communicate with the ‘:Motor((Device))’ object. The ‘:Door((Interface))’ object is used as an interface to communicate with ‘:Door((Device))’ object. The ‘Door((Device))’ and ‘Motor((Device))’ objects simulates the operations of the actual door and motor hardware devices.

Step I: Extract Message-Sequence from the Sequence Diagram

The execution of Step I results in the instantiation of the ‘Message-Sequence’ table as shown Table 5.3. The XMI file of the sequence diagram as shown in Figure 5.3 is used as an input in this step.

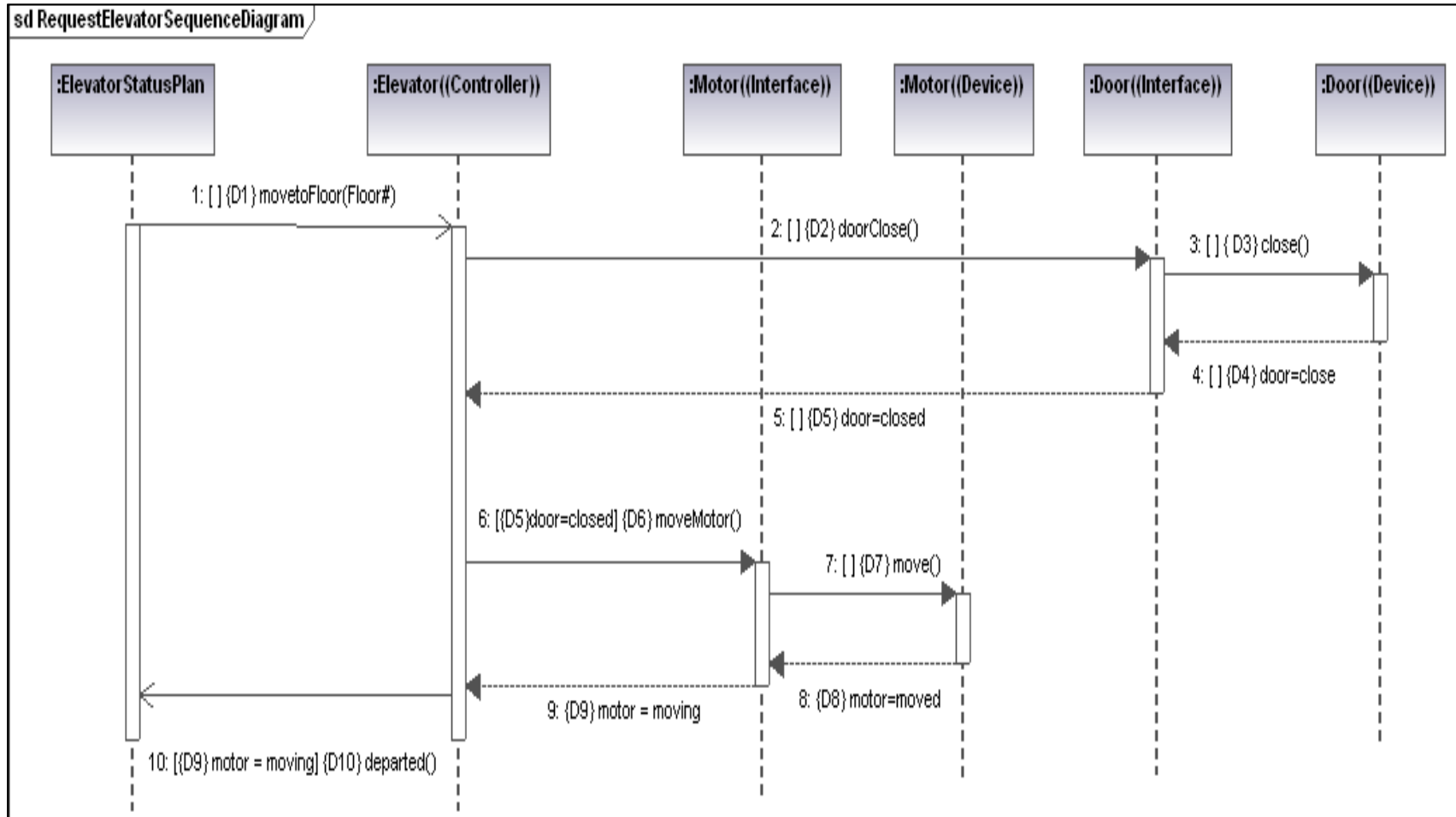


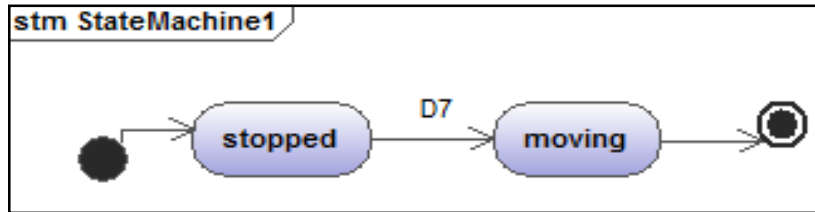
Figure 5.3: Elevator Controller Sequence Diagram (Dispatch Elevator Scenario)

Table 5.3: Message-Sequence Table Generated for Dispatch Elevator Scenario

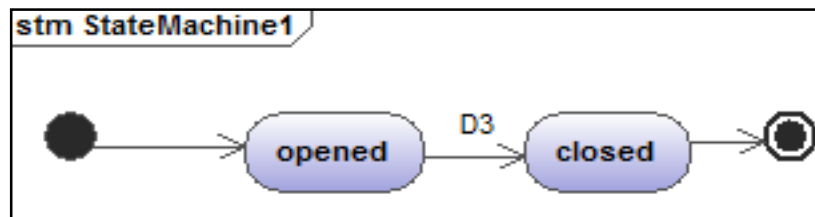
Message#	Name	Type	Pre-condition	From	To	Send/Receive Event Pair
D1	movetoFloor(Floor#)	1		ElevatorStatusPlan	Elevator((Controller))	{E1,E2}
D2	doorClose()	2		Elevator((Controller))	Door((Interface))	{E3,E4}
D3	close()	2		Door((Interface))	Door((Device))	{E5,E6}
D4	door=close	3		Door((Device))	Door((Interface))	{E7,E8}
D5	door=closed	3		Door((Interface))	Elevator((Controller))	{E9,E10}
D6	moveMotor()	2	{D5}door=closed	Elevator((Controller))	Motor((Interface))	{E11,E12}
D7	move()	2		Motor((Interface))	Motor((Device))	{E13,E14}
D8	motor=moved	3		Motor((Device))	Motor((Interface))	{E15,E16}
D9	motor=moving	3		Motor((Interface))	Elevator((Controller))	{E17,E18}
D10	departed()	1		Elevator((Controller))	ElevatorStatusPlan	{E19,E20}

Step II: Identify ‘Event-Sequence-State-Transition’

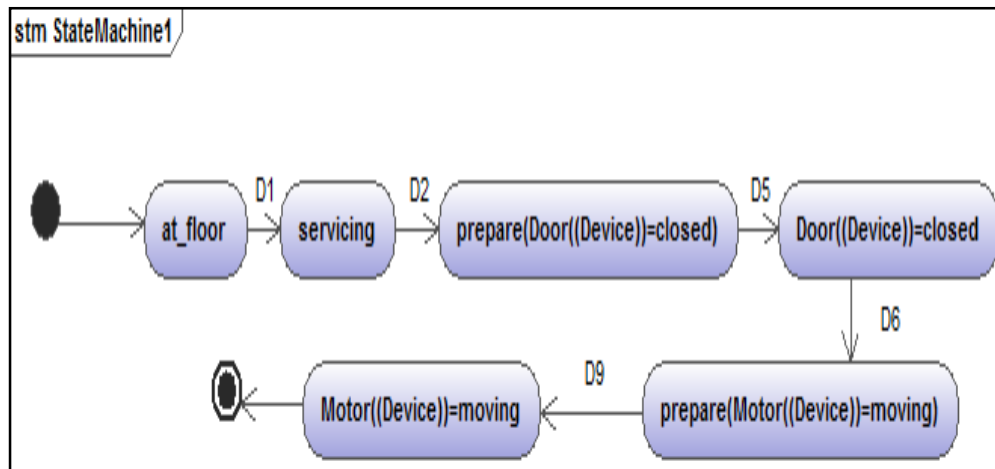
There are three objects, namely *Motor*((Device)), *Door*((Device)) and *Elevator*((Controller)) that change their state during the scenario realization and their state transitions are shown in Figure 5.4(a), Figure 5.4(b) and Figure 5.4(c) respectively.



(a) Motor State Diagram (Dispatch Elevator Scenario)



(b) Door State Diagrams (Dispatch Elevator Scenario)



(c) Elevator Controller State Diagram

Figure 5.4: State Diagrams For Dispatch Elevator Scenario of ECS Application

The initial state of *Elevator*((Controller)) object is ‘*at_floor*’ (i.e., elevator is stationed at floor) and similarly initial state(s) of *Motor*((Device)) and *Door*((Device)) objects are ‘*stopped*’ and ‘*opened*’ respectively.

The message D1 (Figure 5.6) has been sent from the ‘:ElevatorStatusPlan’ object to an ‘:Elevator((Controller))’ object and upon receive of this message the state of the ‘:Elevator((Controller))’ object is changed to ‘servicing’ i.e., state transition occurs during the receive event (event E2 as shown in Table 5.4) of message D1.

The message D2 has been sent from ‘:Elevator((Controller))’ object to ‘:Door((Interface))’ object, but in this case the state of Elevator((Controller)) object is changed to ‘prepare(Door((Device)) = closed)’ immediately after the sending of message D2 i.e. state transition occurs on the send event of message D2 (event E3 of Table 5.6). The state of ‘:Elevator((Controller))’ object is changed to ‘Door((Device)) = closed’ after receiving the reply message D5 (state transition occurs at the receive event of message D5 i.e., at event E10 of Table 5.4). The ‘:Motor((Device))’ object makes a transition from ‘stopped’ to ‘moving’ state after receiving the message D7 and since the sender of message D7 is not ‘:Motor((Device))’, so ‘:Motor((Device))’ object makes its required state transition at the receive event of message D7. Similarly, ‘:Door((Device))’ object makes its required state transition from ‘opened’ to ‘closed’ state after receiving the message D3 i.e., actual state transition occurs at the receive event of message D3.

The ‘Event-Sequence-State-Transition’ table populated after the application of Step II, by taking Table 5.3 and the state diagram XMI files of Figure 5.4 as inputs, is shown in Table 5.4.

Step III and Step IV: Generate Fault Tree XML File and Construct Software Fault Tree

The Case 1 hazardous-state considered for this application is ‘Door((Device))!=closed and Motor((Device))= moving’. The fault tree XML file generated for this Case 1 hazardous-state is shown in Figure 5.5 and the corresponding fault tree generated from this file is shown in Figure 5.6.

The names of basic and intermediate events are to be interpreted as follows:

1. $!(msg\#)$ {where $msg\#$ is the message number} i.e., a corresponding message is not sent whereas it has to be sent
2. $(msg\#)$ {where $msg\#$ is the message number} i.e., a corresponding message is sent, whereas it has not to be sent
3. $^(PreC)$ {where $PreC$ is the precondition of the message} i.e., precondition has been wrongly evaluated as true.

**Table 5.4: Event-Sequence-State-Transition Table Generated for
Dispatch Elevator Scenario**

Event	Type	Timestamp	Elevator((Controller))	Motor((Device))	Door((Device))
E1	1	1	at_floor	stopped	opened
E2	2	2	servicing	stopped	opened
E3	1	3	prepare(Door((Device))=closed)	stopped	opened
E4	2	4	prepare(Door((Device))=closed)	stopped	opened
E5	1	5	prepare(Door((Device))=closed)	stopped	opened
E6	2	6	prepare(Door((Device))=closed)	stopped	closed
E7	3	7	prepare(Door((Device))=closed)	stopped	closed
E8	4	8	prepare(Door((Device))=closed)	stopped	closed
E9	3	9	prepare(Door((Device))=closed)	stopped	closed
E10	4	10	Door((Device))=closed	stopped	closed
E11	1	11	prepare(Motor((Device))=moving)	stopped	closed
E12	2	12	prepare(Motor((Device))=moving)	stopped	closed
E13	1	13	prepare(Motor((Device))=moving)	stopped	closed
E14	2	14	prepare(Motor((Device))=moving)	moving	closed
E15	3	15	prepare(Motor((Device))=moving)	moving	closed
E16	4	16	prepare(Motor((Device))=moving)	moving	closed
E17	3	17	prepare(Motor((Device))=moving)	moving	closed
E18	4	18	Motor((Device))=moving	moving	closed
E19	1	19	Motor((Device))=moving	moving	closed
E20	2	20	Motor((Device))=moving	moving	closed

```
<?xml version="1.0" encoding="UTF-8"?><Fault-Tree><Intermediate-Event><Title>Door((Device))!=closed AND Motor((Device))=moving</Title><And-Gate><Intermediate-Event><Title>Door((Device))!=closed</Title><Or-Gate><Basic-Event><Title>! (D2)</Title></Basic-Event></Or-Gate></Intermediate-Event><Intermediate-Event><Title>Motor((Device))=moving</Title><And-Gate><Basic-Event><Title>^({D5} door=closed)</Title></Basic-Event><Basic-Event><Title>(D6)</Title></Basic-Event><Basic-Event><Title>(D7)</Title></Basic-Event></And-Gate></Intermediate-Event></And-Gate></Intermediate-Event></Fault-Tree>
```

Figure 5.5: faultree.xml File for Case 1 Hazardous-State Door((Device))!=closed and Motor((Device))= moving

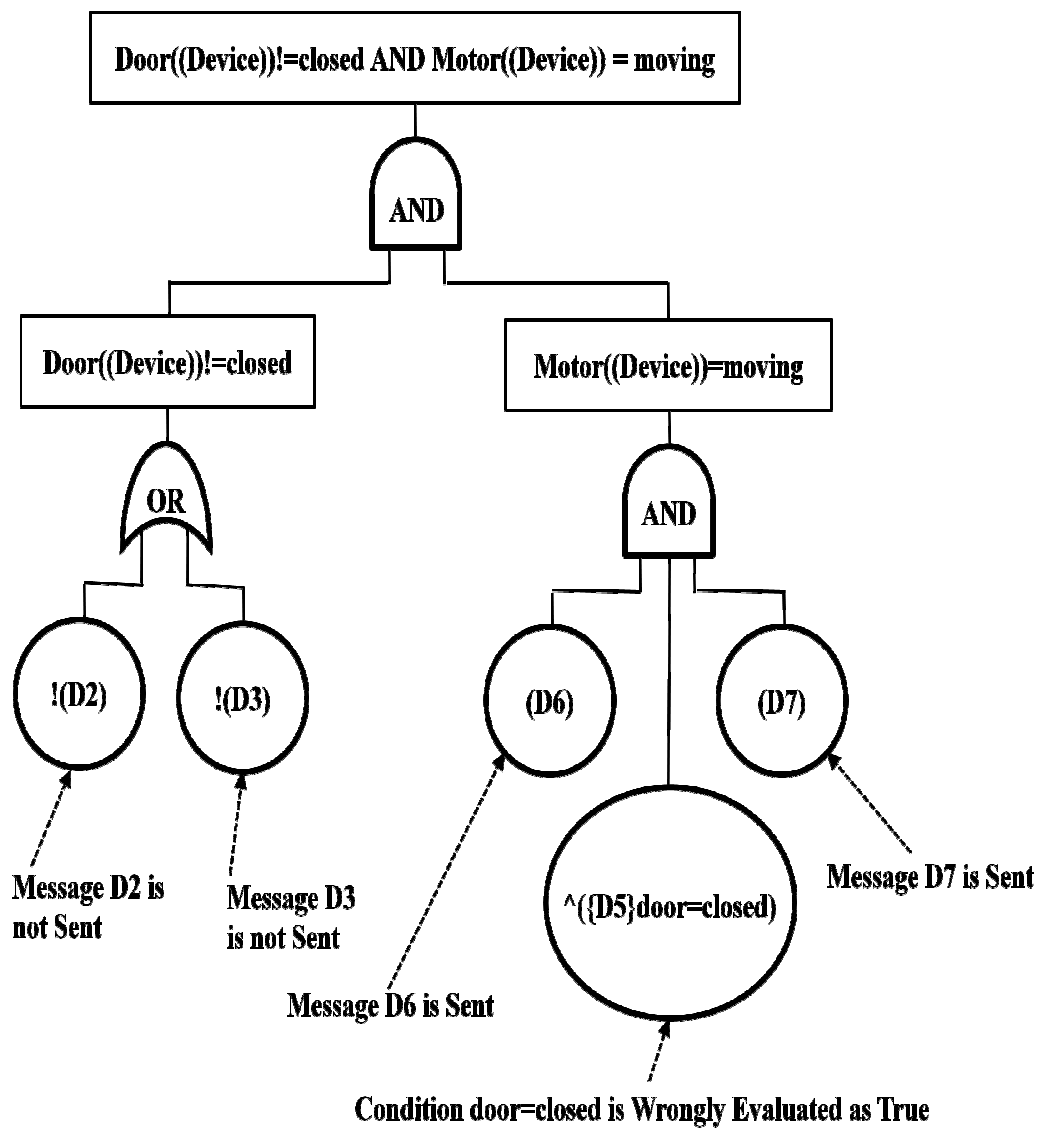


Figure 5.6: Fault Tree Generated for Case 1 Hazardous-State Door((Device))!= closed and Motor((Device)) = moving

Similarly, the *faulttree.xml* file generated for Case 3 hazardous state *Door((Device))!=closed and Motor((Device))!=moving*, is shown in Figure 5.7 and its corresponding fault tree drawn by FaultCAT tool using this file as an input is shown in Figure 5.8.

```
<?xml version="1.0" encoding="UTF-8"?><Fault-Tree><Intermediate-Event><Title>Door((Device))!=closed AND Motor((Device))!=moving</Title><And-Gate><Intermediate-Event><Title>Door((Device))!=closed</Title><Or-Gate><Basic-Event><Title>! (D2)</Title></Basic-Event><Basic-Event><Title>! (D3)</Title></Basic-Event></Or-Gate></Intermediate-Event><Intermediate-Event><Title>Motor((Device))!=moving</Title><Or-Gate><Basic-Event><Title>^( {D5} door=closed)</Title></Basic-Event><Basic-Event><Title>! (D6)</Title></Basic-Event><Basic-Event><Title>! (D7)</Title></Basic-Event></Or-Gate></Intermediate-Event></And- Gate></Intermediate-Event></Fault-Tree>
```

Figure 5.7: faulttree.xml File for Case 3 Hazardous-State Door((Device))!=closed and Motor((Device))!= moving

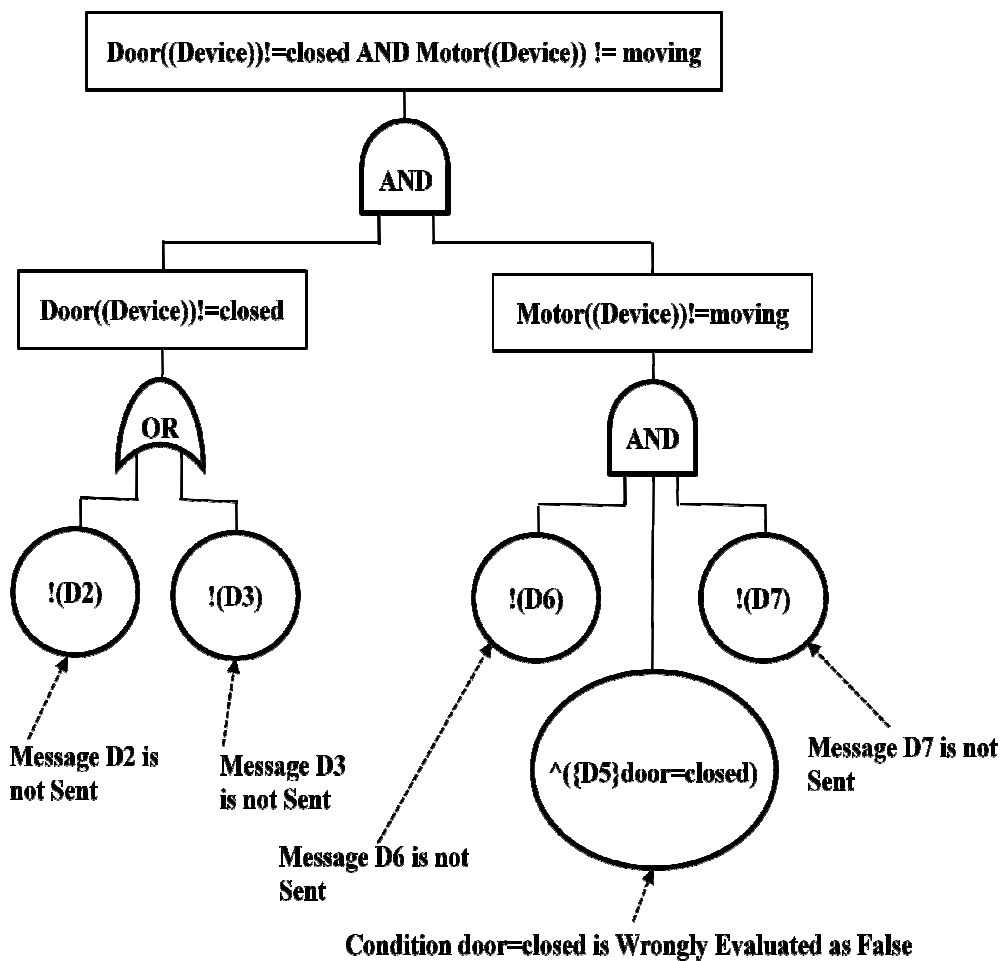


Figure 5.8: Fault Tree generated for Case 3 Hazardous-State Door((Device)) != closed and Motor((Device)) != moving

5.6.2 Stop Elevator Scenario

The ‘Stop Elevator’ use-case functionality is required in the ECS application to stop an elevator at a particular floor number. The sequence diagram drawn for this use-case scenario is shown in Figure 5.9.

The participating objects are ‘:ArrivalSensor((Interface))’, ‘:Elevator((Controller))’, ‘:ElevatorStatusPlan’, ‘:Motor((Interface))’, ‘:Motor((Device))’, ‘:Door((Interface))’, ‘:Door((Device))’ and ‘:DoorTimer’.

The role of the ‘:ArrivalSensor((Interface))’ object is to detect the arrival of the elevator whenever it is approaching a particular floor number and generates an interrupt for the ‘:Elevator((Controller))’ object.

The roles of the ‘:Elevator((Controller))’, ‘:ElevatorStatusPlan’, ‘:Motor((Interface))’, ‘:Motor((Device))’, ‘:Door((Interface))’ and ‘:Door((Device))’ objects are similar to the roles of the same in ‘Dispatch Elevator’ scenario. The role of the ‘DoorTimer’ object is to maintain the time during which the elevator door has to remain open.

The whole functionality gets executed when a ‘floorReach()’ interrupt message is received by the ‘:Elevator((Controller))’ object from an ‘:ArrivalSensor((Interface))’ object. Upon the receipt of the message, the ‘:Elevator((Controller))’ object performs the following tasks in sequence

- (i) It checks with the ‘:ElevatorStatusPlan’ object to know whether an elevator has to stop at that floor number or not.
- (ii) If elevator has to stop at the floor, then following actions are carried out sequentially
 - a. The ‘:Elevator((Controller))’ object commands to stop the motor
 - b. When motor is stopped, the ‘:Elevator((Controller))’ object commands to open the door
 - c. When door is opened, the ‘:Elevator((Controller))’ starts the door timer
 - d. When timer stops, the ‘:Elevator((Controller))’ object checks with the ‘:ElevatorStatusPlan’ object to know about the next floor destination.
 - e. If the elevator is not to visit any other floor then it remains on the last visited floor.

The ‘Message-Sequence’ table populated by taking the sequence diagram of Figure 5.9 as an input is shown in Table 5.5.

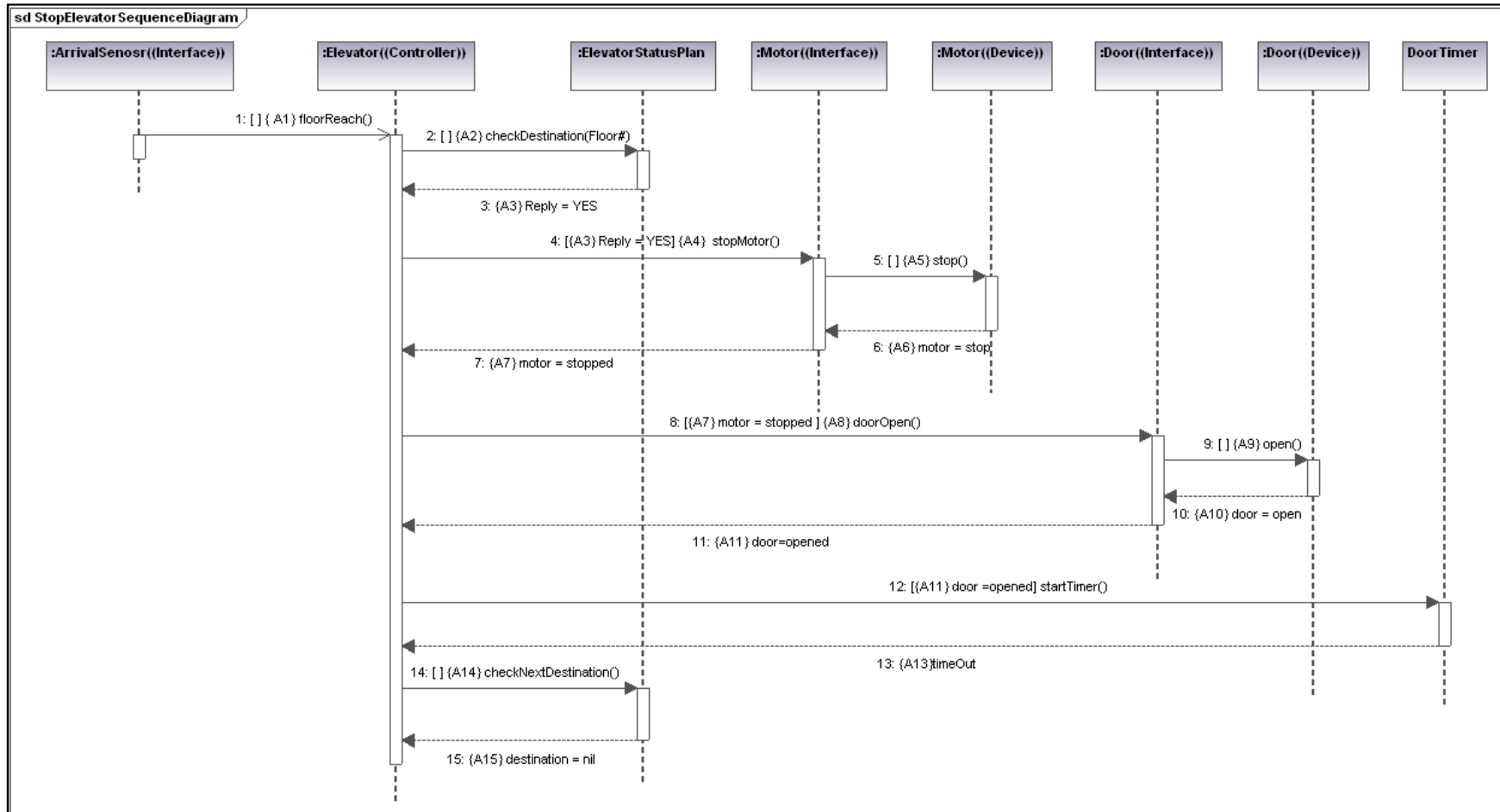
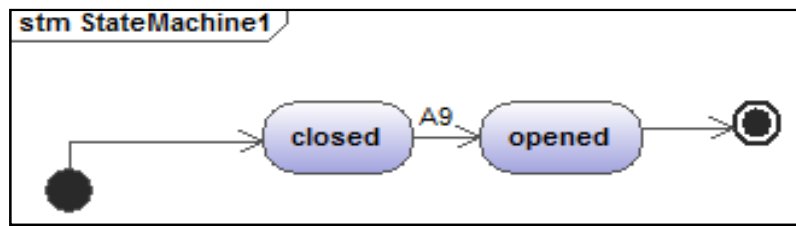


Figure 5.9: Sequence Diagram for Stop Elevator Scenario

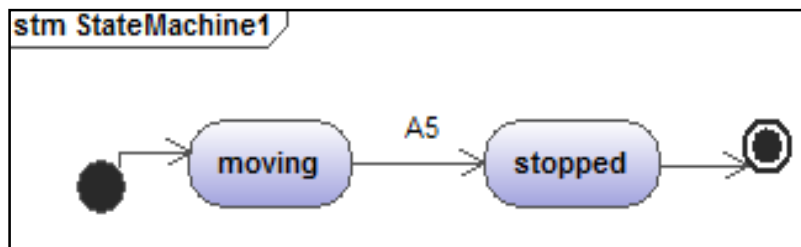
Table 5.5: Message-Sequence Table Generated for Stop Elevator Scenario

Message#	Name	Type	Precondition	From	To	Send/Receive Event-Pair
A1	floorReach(Floor#)	1		ArrivalSensor((Interface))	Elevator((Controller))	{E1,E2}
A2	checkDestination(Floor#)	2		Elevator((Controller))	ElevatorStatusPlan	{E3,E4}
A3	Reply=YES	3		ElevatorStatusPlan	Elevator((Controller))	{E5,E6}
A4	stopMotor()	2	{ A3 }Reply=YES	Elevator((Controller))	Motor((Interface))	{E7,E8}
A5	stop()	2		Motor((Interface))	Motor((Device))	{E9,E10}
A6	motor=stop	3		Motor((Device))	Motor((Interface))	{E11,E12}
A7	motor=stopped	3		Motor((Interface))	Elevator((Controller))	{E13,E14}
A8	doorOpen()	2	{ A7 }motor=stopped	Elevator((Controller))	Door((Interface))	{E15,E16}
A9	open()	2		Door((Interface))	Door((Device))	{E17,E18}
A10	door=open	3		Door((Device))	Door((Interface))	{E19,E20}
A11	door=opened	3		Door((Interface))	Elevator((Controller))	{E21,E22}
A12	startTimer()	2	{ A11 }door=opened	Elevator((Controller))	DoorTimer	{E23,E24}
A13	timeOut	3		DoorTimer	Elevator((Controller))	{E25,E26}
A14	checkNextdestination()	2		Elevator((Controller))	ElevatorStatusPlan	{E27,E28}
A15	destination=nil	3		ElevatorStatusPlan	Elevator((Controller))	{E29,E30}

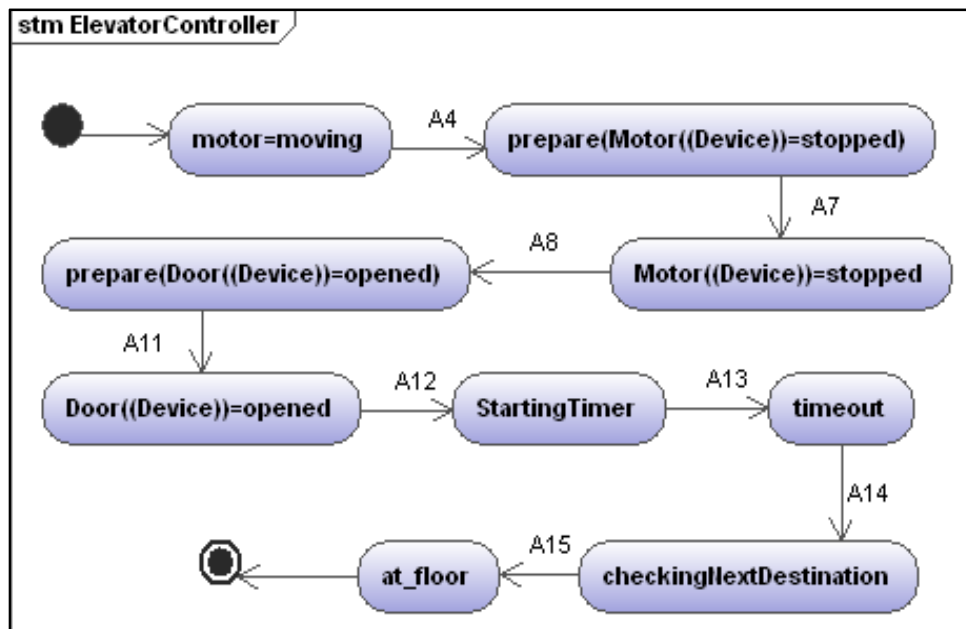
The state diagrams for ‘:Door((Device))’, ‘:Motor((Device))’ and ‘:Elevator((Controller))’ objects are shown in Figure 5.10.



(a) Door State Diagram



(b) Motor State Diagram



(c) Elevator Controller State Diagram

Figure 5.10: State Diagrams for Stop Elevator Scenario

The ‘Event-Sequence-State-Transition’ table populated after the application of Step II of the proposed approach is shown in Table 5.6. The ‘Message-Sequence’ table as shown in Table 5.5 and the XMI files of the state diagrams of Figure 5.10 are used as inputs in this step.

Table 5.6: Event-Sequence-State-Transition Table Generated for Stop Elevator Scenario

Event	Type	Timestamp	Elevator((Controller))	Motor ((Device))	Door ((Device))
E1	1	1	motor=moving	moving	closed
E2	2	2	motor=moving	moving	closed
E3	1	3	motor=moving	moving	closed
E4	2	4	motor=moving	moving	closed
E5	3	5	motor=moving	moving	closed
E6	4	6	motor=moving	moving	closed
E7	1	7	prepare(Motor((Device))=stopped)	moving	closed
E8	2	8	prepare(Motor((Device))=stopped)	moving	closed
E9	1	9	prepare(Motor((Device))=stopped)	moving	closed
E10	2	10	prepare(Motor((Device))=stopped)	stopped	closed
E11	3	11	prepare(Motor((Device))=stopped)	stopped	closed
E12	4	12	prepare(Motor((Device))=stopped)	stopped	closed
E13	3	13	prepare(Motor((Device))=stopped)	stopped	closed
E14	4	14	Motor((Device))=stopped	stopped	closed
E15	1	15	prepare(Door((Deice))=opened)	stopped	closed
E16	2	16	prepare(Door((Deice))=opened)	stopped	closed
E17	1	17	prepare(Door((Deice))=opened)	stopped	closed
E18	2	18	prepare(Door((Deice))=opened)	stopped	opened
E19	3	19	prepare(Door((Deice))=opened)	stopped	opened

Event	Type	Timestamp	Elevator((Controller))	Motor ((Device))	Door ((Device))
E20	4	20	prepare(Door((Deice))=opened)	stopped	opened
E21	3	21	prepare(Door((Deice))=opened)	stopped	opened
E22	4	22	Door((Device))=opened	stopped	opened
E23	1	23	startingTimer	stopped	opened
E24	2	24	startingTimer	stopped	opened
E25	3	25	startingTimer	stopped	opened
E26	4	26	timeout	stopped	opened
E27	1	27	checkingNextDestination	stopped	opened
E28	2	28	checkingNextDestination	stopped	opened
E29	3	29	checkingNextDestination	stopped	opened
E30	4	30	at_floor	stopped	opened

The *faulttree.xml* file generated for Case 1 hazardous state *Motor((Device)) != stopped and Door((Device)) = opened*, is shown in Figure 5.11. The corresponding fault tree drawn by FaultCAT tool using this *faulttree.xml* file as an input is shown in Figure 5.12.

```
<?xml version="1.0" encoding="UTF-8"?><Fault-Tree><Intermediate-Event><Title>Motor((Device))!=stopped AND Door((Device))=opened</Title><And-Gate><Intermediate-Event><Title>Motor((Device))!=stopped</Title><Or-Gate><Basic-Event><Title>^{A3}Reply=YES</Title></Basic-Event><Basic-Event><Title>!(A4)</Title></Basic-Event><Basic-Event><Title>!(A5)</Title></Basic-Event></Or-Gate></Intermediate-Event><Intermediate-Event><Title>Door((Device))=opened</Title><And-Gate><Basic-Event><Title>^{A7}motor=stopped</Title></Basic-Event><Basic-Event><Title>(A8)</Title></Basic-Event><Basic-Event><Title>(A9)</Title></Basic-Event></And-Gate></Intermediate-Event></And-Gate></Intermediate-Event></Fault-Tree>
```

Figure 5.11: Fault Tree XML File for Case 1 Hazardous-State ‘Motor((Device))!= stopped and Door((Device))=opened’

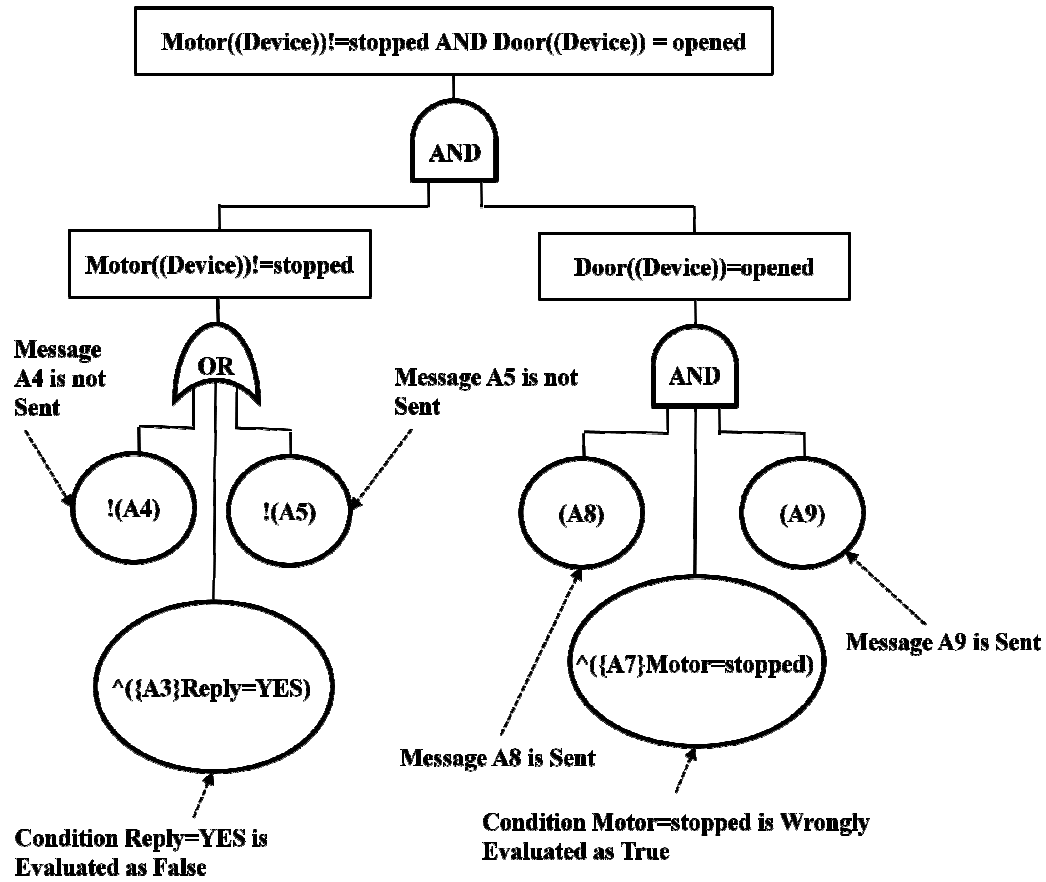


Figure 5.12: Fault Tree Generated for Case 1 Hazardous-State ‘Motor((Device))!= stopped and Door((Device))=closed’

Similarly, the *faulttree.xml* file generated for Case2 hazardous state *Motor((Device))=stopped and Door((Device))!= opened*, is shown in Figure 5.13. The corresponding fault tree drawn by FaultCAT tool using this *faulttree.xml* file as an input is shown in Figure 5.14.

```
<?xml version="1.0" encoding="UTF-8"?><Fault-Tree><Intermediate-Event><Title>Motor((Device))=stopped AND Door((Device))!=opened</Title><And-Gate><Intermediate-Event><Title>Motor((Device))=stopped</Title><And-Gate><Basic-Event><Title>({A3}Reply=YES)</Title></Basic-Event><Basic-Event><Title>(A4)</Title></Basic-Event><Basic-Event><Title>(A5)</Title></Basic-Event><Basic-Event><Title>(A6)</Title></Basic-Event><Basic-Event><Title>(A7)</Title></Basic-Event></And-Gate></Intermediate-Event><Intermediate-Event><Title>Door((Device))!=opened</Title><Or-Gate><Basic-Event><Title>^({A7}motor=stopped)</Title></Basic-Event><Basic-Event><Title>!(A8)</Title></Basic-Event><Basic-Event><Title>!(A9)</Title></Basic-Event></Or-Gate></Intermediate-Event></And-Gate></Intermediate-Event></Fault-Tree>
```

Figure 5.13: faulttree.xml File for Case 2 Hazardous-State Motor((Device)) = stopped and Door((Device)) != opened

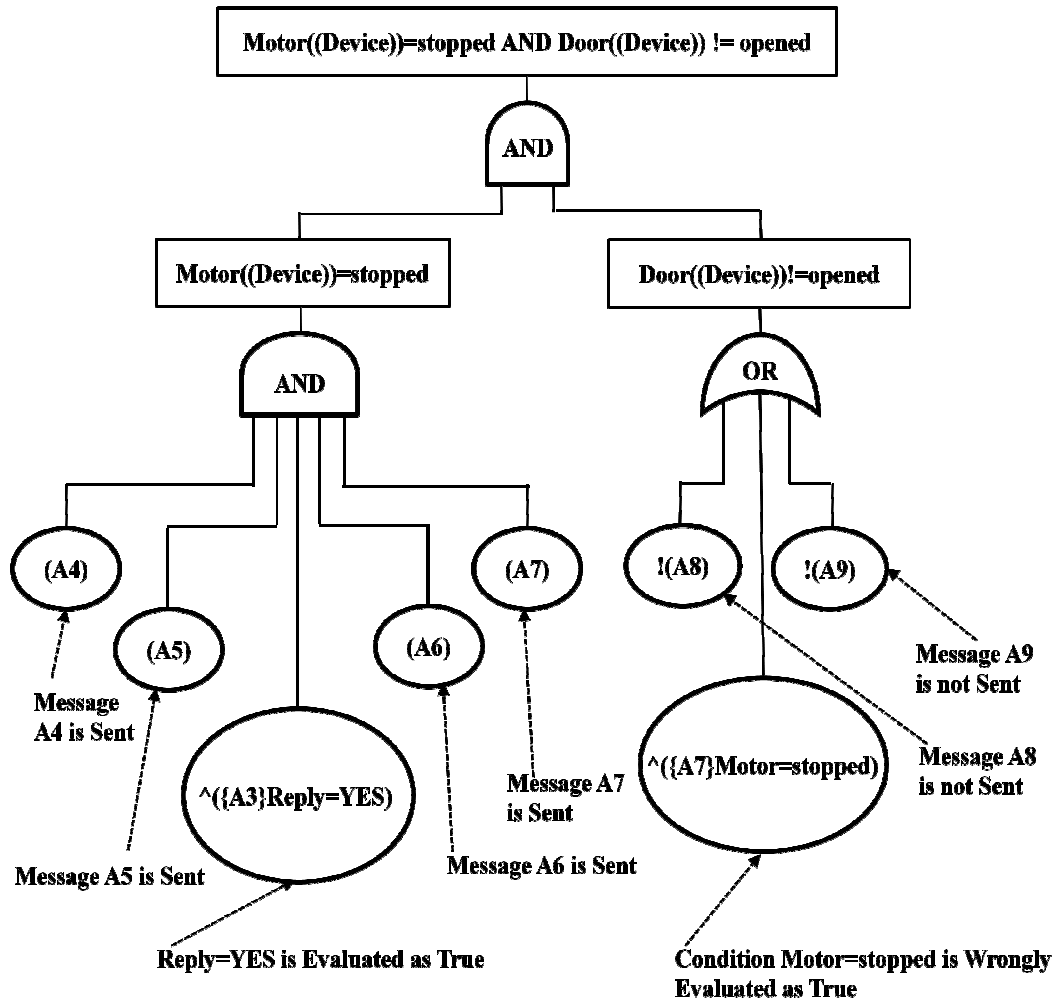


Figure 5.14: Fault Tree Generated for Case 1 Hazardous-State Motor((Device)) = stopped and Door((Device)) != closed

The snapshots of the fault trees constructed using a FaultCAT tool, from the faulttree.xml files as shown in Figure 5.5, Figure 5.7, Figure 5.11 and Figure 5.13, are also shown in Appendix-V, Appendix-VI, Appendix-VII and Appendix-VIII, respectively.

5.6.3 Analysis of Results

The aim of the SFTA algorithm is to construct the fault trees for the selected hazardous-state of the system. The analyst has to incorporate the necessary safety features in the system to avoid the occurrence of the hazardous state. The analyst uses the cutset analysis results of the fault tree for this purpose. The cut sets are the logical combinations of message-related errors that cause the hazardous-state. A cut set is a minimal cutset if none of the events from a cutset can be removed and a hazardous-state can still occur. A

hazardous-state can be avoided by providing safeguards against the selected erroneous events from the minimal cut sets. If the events in the minimal cut sets are joined by an 'AND' gate, then the hazardous state can be avoided by providing the safety features for any one erroneous event selected from the list. The experience and the domain knowledge of the analyst plays an important role in the selection of the erroneous event in this situation.

Consider the fault tree for the hazardous-state 'Door((Device))!=closed AND Motor((Device))=moving' as shown in Figure 5.6. There are two minimal cut sets for this fault tree and these are {'!(D2)', '(D6)', '^({D5}door=closed)', '(D7)'} and {'!(D3)', '(D6)', '^({D5}door=closed)', '(D7)'}. In the both these minimal cut sets, the erroneous events {'(D6)', '^({D5}door=closed)', '(D7)'} are joined by an 'AND' gate. So any one of these can be selected for the avoidance of the hazardous-state. If the erroneous event '^({D5}door=closed)' is selected from these events, then to avoid the hazardous-state 'Door((Device))!=closed AND Motor((Device))=moving' from occurring, the analysts has to provide the necessary safeguards against the three erroneous events namely '!(D2)', '!(D3)' and '^({D5}door=closed)'.

Similarly, consider the fault tree for the hazardous-state 'Motor ((Device))!= stopped and Door((Device)) = opened' as shown in Figure 5.12. There are three minimal cut sets for this fault tree and these are {'!(A4)', '(A8)', '^({A7}Motor=stopped)', '(A9)'}, {'({A3}Reply=YES)', '^({A7}Motor=stopped)', '(A8)', '^({A7}Motor=stopped)', '(A9)'} and {'!(A5)', '(A8)', '^({A7}Motor=stopped)', '(A9)'}. The erroneous events {'(A8)', '^({A7}Motor=stopped)', '(A9)'} in all the three minimal cut sets are joined via an 'AND' gate. If an erroneous event '^({A7}Motor=stopped)' is selected from this list then to avoid the hazardous-state 'Motor((Device)) != stopped and Door((Device)) = opened' from occurring the safeguards have to be provided against four erroneous events and these are '!(A4)', '({A3}Reply=YES)', '!(A5)' and '^({A7}Motor=stopped)'.

5.7 COMPARATIVE ANALYSIS

The presented SFTA approach can be considered as an extension of the manual SFTA application work described by Massood et al. (2002, 2003) where only the basic guidelines, to convert a given sequence and state diagram to its corresponding software fault tree, have been described. The message type errors such as 'message not sent' or 'message sent at a wrong time' have to be identified manually whereas in our approach,

the message errors are identified by software and fed during construction of the software fault tree. The focus of the work presented by Pai and Dugan (2002) is reliability assessment and the reliability related information such as redundancy; reconfiguration and dependencies among components have been embedded in the architectural model itself. The UML class has been used to model the redundancy, whereas UML activity chart is used to model the failure behavior. In UML, class diagrams are generally used to represent the static structure/components of the system, whereas our approach is based upon sequence and state diagram(s) which are actually used to express the dynamic behavior of the system. Lauer and German's work (2011) also focused upon reliability assessment. The focus of the presented semi-automated SFTA approach is the hazard analysis so that the analyst could analyze how a particular hazard can occur in the system. The procedure for embedding the features of FTA and one other hazard analysis technique named hazard and operability (HazOp) into UML component models has been described by Lu et al. (2005) and in this case also the application process is manual. The fault tree construction step of the presented SFTA approach is automated and error-free.

The shortcomings of the proposed approach are:

- 1 It can be applied to a hazardous-state that can be expressed in terms of an incompatible state(s) of the collaborating objects. In actual real time applications hazard may occur because of many other factors such as wrong computation, incorrect response received etc.
- 2 The sequence diagram is drawn for a single scenario without the use of 'alt' blocks. But, a given use case functionality can have many different scenarios as discussed in Chapter 3
- 3 The proposed approach requires that the tagging of sequence diagram should be proper and correct. If any message's precondition tag is specified wrongly, then the precondition basic error event and the constructed fault tree will also be wrong. If any message's precondition tag is missing, then there will not be any basic event representing the precondition of that message in the final constructed fault tree. Another type of errors such as 'wrong assignment of the message numbers' or 'not assigning the message numbers' for various messages in the sequence diagram, will affect the final generated fault tree.

Work on these shortcomings is carried on as further work.

Software Failure Modes and Effects Analysis in Object-Oriented Design Phase

This chapter presents a new automated SFMEA approach for object-oriented design phase. The proposed approach is of a forward analysis type, which investigates the effects of various message-related errors on the system. The message-related errors are selected from the sequence diagram whereas the effects of these errors are traced to the erroneous states of the participating objects. The approach has been validated by applying it on two safety-critical applications, namely Insulin Delivery System (IDS) and Railway Track Door Control System (RTCS) discussed in previous chapters. The UML sequence diagram and state diagrams are required as inputs in XMI (XML Metadata Interchange) format. The Altova UML (Altova-UModel, 2014) tool is used first to draw the required sequence and state diagrams and then export the drawn diagrams to XMI format.

6.1 MOTIVATION FOR SFMEA IN OBJECT-ORIENTED DESIGN PHASE

Guiochet and Baron (Guiochet and Baron, 2003) have applied Failure Modes Effects and Criticality Analysis (FMECA) technique on UML sequence diagrams by identifying eleven types of message-related errors that can occur in the system. The work of Hecht and Hecht (Hecht and Hecht, 2004) described a computer-aided SFMEA approach for two stages of software development namely concept phase and design/implementation phase. Ozarin (Ozarin, 2004) recommended applying Software FMEA approach during the full software life cycle by exploiting various UML diagrams as inputs. The SFMEA approach described by David (David et al, 2008) generates a FMEA table from a sequence diagram, but requires that dysfunctional behaviors of various classes in the form of a database are to be known in advance. It is to be noted that none of the researchers have proposed a solution for SFMEA in object-oriented design phase. The proposed SFMEA approach is an attempt to integrate and automate the application of the SFMEA in object-oriented design phase. The proposed approach also overcomes the following two limitations of the SFTA approach developed and discussed in Chapter 5.

- (i) The sequence diagram can have 'alt' block and can have many scenarios.
- (ii) There is no restriction on naming the states of the participating components.

6.2 OVERVIEW OF THE APPROACH

The proposed SFMEA approach is divided into four steps to investigate the critical effects of various message-related errors on the system. The UML sequence diagram (drawn for the selected use-case functionality) and the UML state diagrams (drawn for the selected collaborating objects) are the required inputs and each of these inputs are accepted in a machine-readable format. The proposed SFMEA approach does a forward analysis of various message-related errors to find out the hazardous-state level effects caused because of these errors.

The first step performs the following two tasks

- (i) It converts a sequence diagram to its pseudo code form, and
- (ii) It extracts the attributes of each message and stores the results as '*Message-Details*'.

The attributes that are extracted for each message and the meaning of each attribute are shown in Table 6.1.

Table 6.1: Various Attributes of a Message

Message Attribute	Meaning
Message#	Unique Message Number assigned to each Message
Message-Name	Name of the message as used in the Sequence Diagram
Label	Label assigned to each message
From	Name of the Sender Object
To	Name of the Receiver Object
Type	Type of Message: 1 for synchronous type and 2 for other type
isReply	Is the message a reply type or not: 1 for reply type and 0 otherwise
Reply-Message	Name of the reply message of a send type message, if any
Send Event	Send Event of the Message
Receive Event	Receive Event of the Message
Message-Send-Dependency-List	List of messages that are dependent upon the message
Message-Send-Independent-List	List of messages that are not dependent upon the message

A typical sequence diagram can have any number of uniquely executable paths known as *scenarios* and each such *scenario* has its own associated message-sequence. The second step extracts the message-sequence for each *scenario*. The message-sequence of any scenario has the structure as shown in Table 6.2.

Table 6.2: Structure of Message-Sequence

Message#	Label	Precondition	Sequence-No
<<Unique Message Number assigned to each message>>	<<Label of each message >>	<<Precondition that must be true before sending the message>>	<<Sequence number of the message in the scenario>>

The third step maps the events of various messages against the states of the collaborating objects. The results are stored in the form of ‘Event-Sequence-State-Transitions’. The structure of ‘Event-Sequence-State-Transition’ for any scenario depends upon the number of objects for which state diagrams are supplied as inputs. If the state diagrams are supplied for two objects, namely ‘X’ and ‘Y’ then the structure of ‘Event-Sequence-State-Transition’ will have four fields as shown in Table 6.3.

Table 6.3: Structure of Event-Sequence-State-Transition

Event#	Logical-Time	X	Y
<<Unique Event Number assigned to each event>>	<<Time of occurrence of the event in a scenario>>	<<State of object X during the event>>	<<State of object Y during the event>>

The ‘Message-Errors-Effects-Analysis’ of each scenario is carried-out in the last step. The fourth step first identifies the various ‘Message-Related’ errors that a system can experience and then investigates the state level effects of these errors on the system. The results of this step, for each scenario, are stored in a tabular form known as ‘Message-Errors-Effects-Analysis’. The structure of the ‘Message-Errors-Effects-Analysis’ table has three fields, namely (i) Message#, (ii) Message-Error and (iii) Effects as shown in Table 6.4. The ‘Effects’ column of Table 6.4 is further sub-divided into various events sub-columns and the number of these events sub-columns depend upon the number of events in the associated scenario.

Table 6.4: Structure of Message-Errors-Effects-Analysis

Message#	Message-Error	Effects		
		Event-1	...	Event-N
<<Message number assigned to each message>>	<<An error that can occur in the message>>	<<Effects of the error observed during various events>>		

An overview of all the four steps of the approach is shown in Figure 6.1.

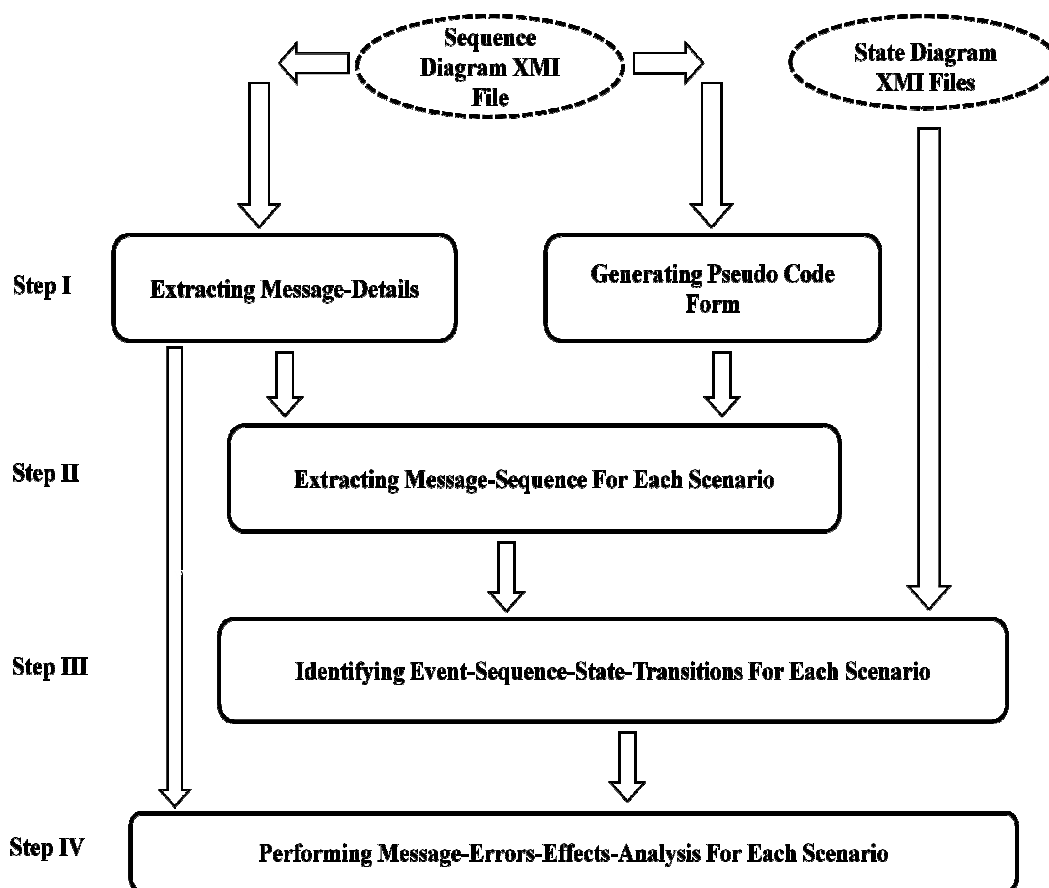


Figure 6.1: Overview of the Proposed SFMEA Approach

6.3 THE PROPOSED SFMEA ALGORITHM

In order to explain proposed SFMEA algorithm in the following sections, a simple sequence diagram showing the interaction among four objects, namely 'A', 'B', 'C' and 'D', as shown in Figure 6.2, is used as an example. This sequence diagram is drawn using the Altova UML (Altova-UModel, 2014) tool. This tool provides a feature where each message is assigned a number known as a sequence number. This sequence number is extracted as the value of 'Label' field (see Table 6.1). The Altova UModel tool provides the support for two types of numbering schemes, namely (i) nested numbering scheme and (ii) simple numbering scheme. In the nested numbering scheme, the sequence number is assigned only to send type of messages, whereas no sequence number is assigned to reply type messages. Each send type message is assigned a sequence number by embedding the sequence number of the message which has activated the current interaction. This numbering scheme is shown in Figure 6.2. The message 'M1(){1}' is assigned a sequence number as '1', the message 'M2(){1.1}' is assigned a sequence number as '1.1' because the interaction is started by a message 'M1()' with sequence number as '1'. In the nested numbering scheme, no sequence number is assigned to reply type messages.

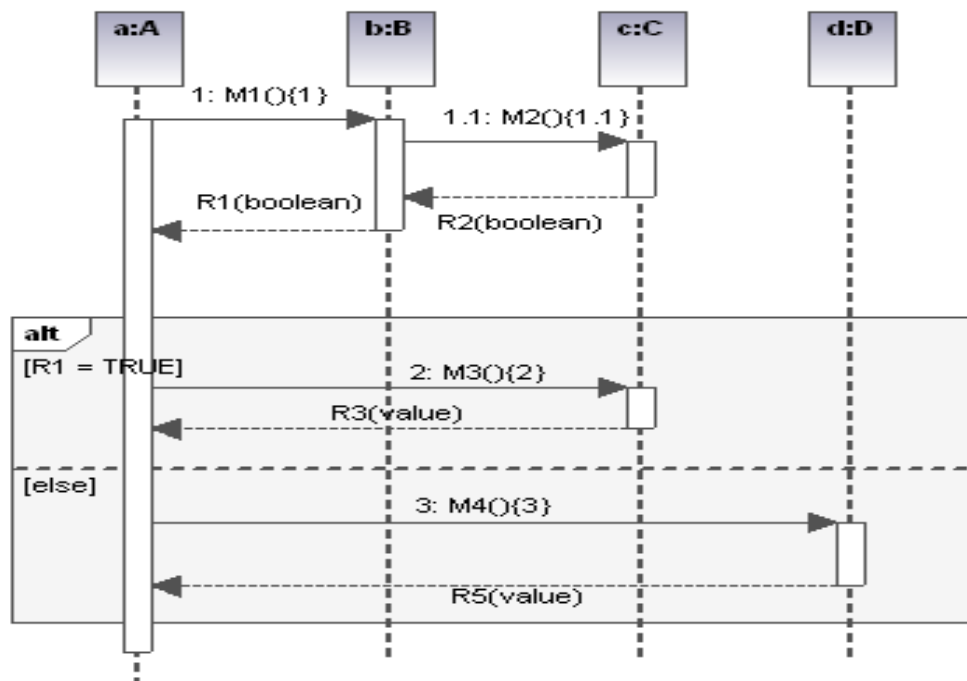


Figure 6.2: A Simple Sequence Diagram with Nested Numbering Scheme

The proposed approach requires that the sequence diagram is to be drawn using the nested numbering scheme because it helps in the identification of the dependent messages. The proposed approach also requires that the sequence number assigned by the tool is to be embedded at the end of the name of a send type message using the syntax ‘{‘x’}’ where ‘x’ is the sequence number assigned by the tool. For example, the message ‘M2(){1.1}’ is embedded with sequence number 1.1. The sequence numbers assigned by the tool are used as ‘Label’ in the proposed approach.

6.3.1 Step I: Generating Pseudo Code Form of Sequence Diagram and Extracting Message-Details

Two tasks are carried out in this step, namely (i) Generating a pseudo code equivalent of the sequence diagram XMI file and (ii) Extracting the attributes of each message. These tasks are described in detail in the following sections.

Step I(a) Generating pseudo code equivalent of the sequence diagram XMI file

A pseudo code form of a sequence diagram contains the names of various messages and the sequence in which these messages are sent. The pseudo code equivalent forms of three UML interaction operators are shown in the Table 6.5. The meaning of the ‘alt’, ‘opt’ and ‘break’ interaction operators is discussed in Chapter 1 and more information about these interaction operators can be found in the work by Booch (Booch et al, 2005).

Table 6.5: Pseudo Code Forms of Various Interaction Operators

Interaction Operator	Pseudo code Equivalent Form	Meaning
‘alt’ block	IF condition THEN <Messages-List-1 is to be sent> ELSE <Message-List-2 is to be sent> ENDIF	‘alt’ stands for alternative flow of actions. There are two paths and any one is to be followed at runtime. IF condition with an ELSE option.
‘opt’ block	IF condition THEN <<Messages-List is to be sent> ENDIF	‘opt’ block stands for optional block i.e.an IF option without an ELSE part. If condition is true then this optional block is executed otherwise it will be skipped.
‘break’ block	IF condition THEN <Messages-List is to be sent> EXIT ENDIF	If condition is true, then break sequence is followed and after that, the functionality is exited. ‘EXIT’ means exit from the functionality and all successive messages are skipped

The execution of the step results in a single text file, which represents the pseudo code equivalent of the given sequence, diagram.

The pseudo code description of this step is given below

<i>Procedure</i>	<i>generatePseudocode()</i>
<i>Input</i>	<i>Sequence Diagram XMI File</i>
<i>Output</i>	<i>Pseudo Code Form Text File</i>

1. *Create a pseudo code form Text File*
2. **FOR** *each message tag in the sequence diagram XMI file*

Case: 'alt' block

- (i) *Write the pseudo code equivalent form into the Text File as per Table 6.5*
- (ii) *Identify messages in the true block and write message names in the Text file*
- (iii) *Write 'ELSE' in the Text File*
- (iv) *Identify messages in the false block write message names in the Text file*
- (v) *Write 'ENDIF' in the Text File*

Case: 'opt' block

- (i) *Write the pseudo code equivalent in to the Text File as per Table 6.5*
- (ii) *Identify messages in the block and write message names in the Text file*
- (iii) *Write 'ENDIF' in the Text File*

Case: 'break' block

- (i) *Write the pseudo code equivalent in to the Text File as per Table 6.5*
- (ii) *Identify messages in the block and write message names in the Text file*
- (iii) *Write 'EXIT' and then 'ENDIF' in the Text File*

Case: a normal message tag

Write the name of the message in the Text File

ENDFOR

The algorithm applied to the sequence diagram of Figure 6.3(a) generates the pseudo code form as shown in Figure 6.3(b). It can be observed that the message 'M3()' with sequence number value as '2' and the message 'R3(value)' are sent only if the condition 'R1=TRUE' is satisfied otherwise the messages 'M4()' and 'R5(value)' are sent. These two alternative sequences of messages are represented by using IF..THEN..ELSE..ENDIF construct in the corresponding pseudo code form in Figure 6.3(b).

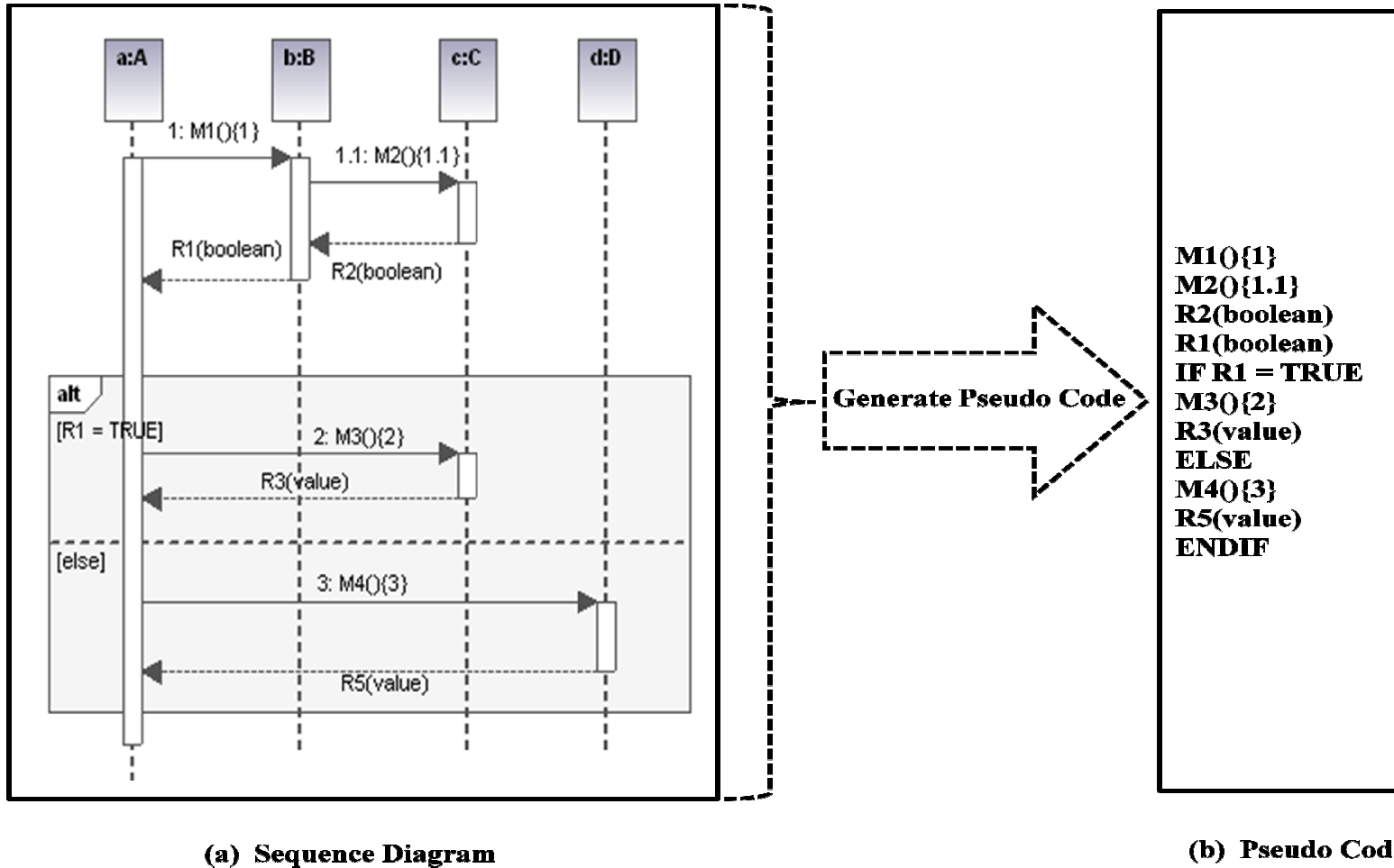


Figure 6.3: Generating Pseudo Code Form from Sequence Diagram

Step I(b) Extracting ‘Message-Details’

The objective of this step is to extract the attributes of the messages by taking a sequence diagram XMI file as an input. The extracted attributes are stored as a table named ‘*Message-Details*’ with a structure as shown in Table 6.1. Three main tasks are carried out in this step and these are as follows.

- (i) A unique message number (Message#) is assigned to each message
- (ii) A unique send and receive event pair is generated for each message
- (iii) Two separate lists, namely ‘Message-Send-Dependency-List’ and ‘Message-Send-Independent-List’ are computed for each send type message.

The ‘Message-Details’ extracted by taking a sequence diagram of Figure 6.2 as an input is shown in Table 6.6. The ‘Message-Details’ as shown in Table 6.6 are extracted as follows:

The name of the message used in the sequence diagram embeds the values of the ‘Message-Name’ and the ‘Label’ fields. The information about ‘From’, ‘To’ and ‘Reply-Message’ fields is contained in the sequence diagram XMI file and is extracted accordingly. The values of the ‘Type’ and the ‘isReply’ fields are assigned as mentioned in Table 6.1. A unique message number (‘Message#’) is assigned for each message. Each message is associated with two events, namely ‘Send Event’ and ‘Receive Event’. The ‘Send Event’ is associated with the sender object of the message, whereas the ‘Receive Event’ is associated with the receiver object of the message. A unique ‘Send Event’ and a ‘Receive Event’ is generated for each message. The procedure for assigning the message numbers and generating the event numbers is identical to the procedure used in the SFTA approach discussed in Chapter 5. The value for the ‘Send-Message-Dependency-List’ and the ‘Send-Message-Independent-List’ fields are computed as follows:

A message ‘M1’ is in the ‘Send-Message-Dependency-List’ of message ‘M2’ if the ‘Label’ field value of the message ‘M1’ contains the ‘Label’ field value of the message ‘M2’. A message ‘M1’ is in the ‘Send-Message-Independent-List’ of message ‘M2’ if the ‘Label’ field value of the message ‘M1’ does not contain the ‘Label’ field value of the message ‘M2’ and the message ‘M1’ is sent after the message ‘M2’ by the same sender object. It is to be noted that only the values of the ‘Label’ fields are included in both the lists.

Consider the sequence diagram as shown in Figure 6.3(a). The message 'M2()' is dependent upon the message 'M1()' because the 'Label' field value of the message 'M2()' is '1.1' and it contains the 'Label' field value of the message 'M1()' which is '1'. The messages 'M3()' and 'M4()' are not dependent upon the message 'M1()' because of the following reasons:

- (i) The 'Label' field values of the messages 'M3()' and 'M4()' are '2' and '3' respectively, and these values do not contain the 'Label' field value of the message 'M1()' which is '1', and
- (ii) The messages 'M3()' and 'M4()' are sent after the message 'M1()' by the same sender object named 'a:A'.

The 'Label' field value of the message 'M2()' is '1.1' and this value is not contained in any other message's 'Label' field value and no message is sent after 'M2()' by the sender of 'M2()' i.e. the object 'b:B'. That's why the values of 'Send-Message-Dependency-List' and the 'Send-Message-Independent-List' fields for the message 'M2()' are blank.

Similarly, consider the message 'M3()' with 'Label' value as '2'. There exists no message in the sequence diagram of Figure 6.3(a) whose 'Label' field value starts with '2'. So, the 'Send-Message-Dependency-List' of the message 'M3()' does not have any message. But the message 'M4()' with 'Label' value as '2' appears in the 'Send-Message-Independent-List' of this message because the message 'M4()' is sent after the message 'M3()' by the same sender object 'a:A'.

Table 6.6: Message-Details Extracted from Sequence Diagram of Figure 6.2

Message#	Message-Name	Label	From	To	Type	isReply	Reply-Message	Send Event	Receive Event	Message-Send-Dependency-List	Message-Send-Independent-List
A1	M1()	1	A	B	1	0	R1(boolean)	e1	e2	1.1	2,3
A2	M2()	1.1	B	C	1	0	R2(boolean)	e3	e4		
A3	R2(boolean)	M2()	C	B	1	1		e5	e6		
A4	R1(boolean)	M1()	B	A	1	1		e7	e8		
A5	M3()	2	A	C	1	0	R3(value)	e9	e10		5
A6	R3(value)	M3()	C	A	1	1		e11	e12		
A7	M4()	3	A	D	1	0	R5(value)	e13	e14		
A8	R5(value)	M4()	D	A	1	1		e15	e16		

The pseudo code of algorithm for this step is given below

Procedure *ExtractMessageDetails()*
Input *Sequence Diagram XMI File*
Output *Message-Details*

FOR each message tag in the sequence diagram XMI file

- (i) Extract values of Message-Name, Label, From, To, isReply, Reply-Message Fields from the XMI file
- (ii) Compute the values of the 'Type' and the 'isReply' fields as per Table 6.1
- (iii) Assign a Unique Message# to each message
- (iv) Generate a unique send event and receive event for each message
- (v) Compute the values for the Message-Send-Dependency-List and the Message-Send-Independent-List for each send type of message

ENDFOR

6.3.2 Step II: Extracting Message-Sequence for each Scenario

The presence of each 'alt', 'opt' or 'break' block in the sequence diagram represents the existence of two possible scenarios and each such scenario has its own message sequence. For example, the sequence diagram of Figure 6.3(a) has two scenarios with message-sequences as: (i) {M1(), M2(), R2(boolean), R1(boolean), M3(), R3(value)} and (ii){M1(), M2(), R2(boolean), R1(boolean), M4(), R5(value)}. The objective of this step is to extract message-sequence for each scenario.

This step takes the pseudo code form of the sequence diagram and the 'Message-Details' generated in the first step as inputs and constructs a graph named 'Message-Sequence-Control-Flow-Graph' (MSCFG) using only the message numbers. The MSCFG is a collection of nodes where a node belongs to either a normal node or a conditional node. The normal node contains two parts, namely node-details and next-pointer. The node-details part gives the information about the name of the message as used in the pseudo code text file. The next-pointer field contains the address of the node where a control is transferred from the current node. The conditional node has a conditional expression associated with it and contains three fields, namely (i) node-details, (ii) true-next-pointer and (iii) false-next-pointer. The node-details part gives the information about the name of the message and a conditional expression associated with the node. The true-next-pointer field points to the node where the control is transferred in case the result of the conditional expression associated with the node is true. Similarly, the false-next-pointer field points to the node where the control is transferred in case the result of the conditional expression associated with the node is false. The MSCFG is actually

constructed from the pseudo code text file and later on each individual message is replace the corresponding message number from the 'Message-Details' table.

The MSCFG constructed by taking the pseudo code description as shown in Figure 6.3(b) and the 'Message-Details' as shown in Table 6.6 as inputs, is shown in Figure 6.4. The nodes A1,A2,A3 are normal nodes, whereas the node A4 is a conditional node. The control from the A4 is shifted to node A5 only if the condition R1=TRUE is satisfied otherwise the control is shifted to node A7.

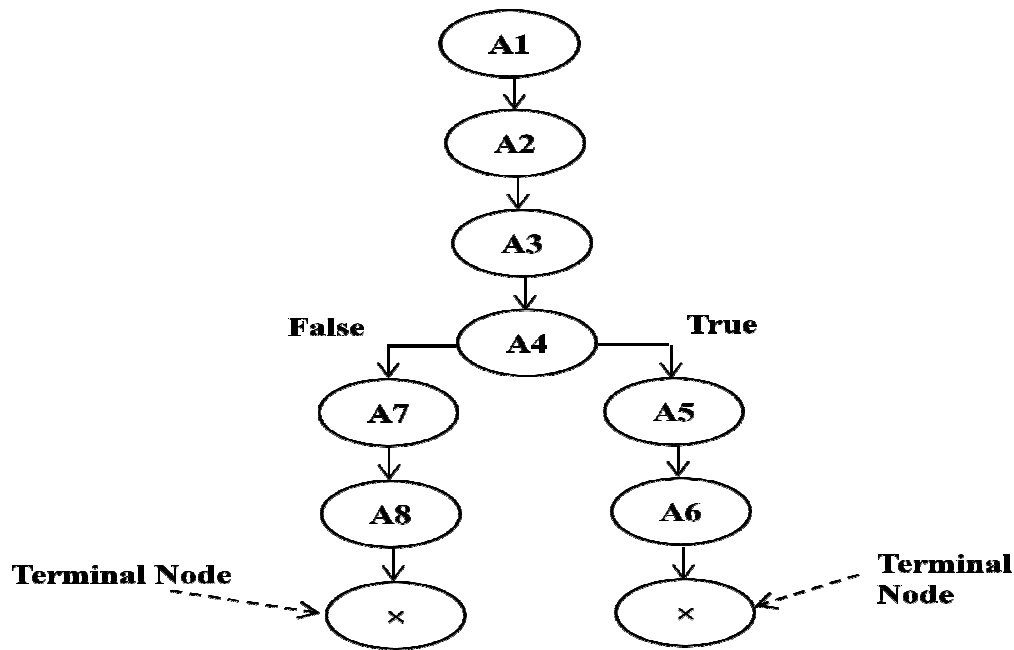


Figure 6.4: Message-Sequence-Control-Flow-Graph Constructed For Pseudo Code Description of Figure 6.3(b)

The pseudo code for the construction of the 'MSCFG' is given below:

```

Procedure    constructMSCFG()
Input       Pseudo code Text File and "Message-Details"
output      Message-sequence Control Flow Graph

Node  startT=null    /* start node */
Node  last=null     /* terminal node */
Stack messageStack  /* message stack of the messages */
Stack ifStack       /* messages in the true part of IF block */
Stack elseStack     /* messages in the false part of If block */

```

```

FOR each message-line of the input file
  read message-line from the input file;
  create Node temp=null,curr=null,next=null;
  IF message-line is not null THEN

```

```

create new Node 'n1' with description as message-line;
set next = n1;
IF start is null THEN
    set start = next;
ENDIF
switch(next)
{
    case: next is "IF"
    {
        (i) set true part of the CURR node to NEXT;
        (ii) set latest event as CURR in eventStack;
        (iii) set latest IF event as CURR in ifStack;
    } // End of case
    case: next is "ELSE"
    {
        (i) set next event of curr as next;
        (ii) IF latest event in messageStack is IF type THEN
            set false part of latest IF event as curr
            ELSE
                set false part of latest ELSE event as curr
            ENDIF
        (iii) pop out the latest element from messageStack;
        (iv) pop out latest element from elseStack;
    } // End of case
    case: next is "ENDIF"
    {
        (i) pop out elements from messageStack as there are 'ELSE' type
            events since the last 'IF' type event;
        (ii) remove the last 'IF' type event from messageStack;
        (iii) IF 'ENDIF' type event occurs right after 'IF' type THEN
            set false part of IF type to curr
            ELSE
                remove latest IF type event from ifStack
            ENDIF
        IF next is 'ENDIF' or 'ELSE' type THEN
            push curr into messageStack
        ENDIF
    } // End of case
    case: next is normal message
    {
        IF next is 'ENDIF' type or 'ELSE' THEN
            push curr into messageStack
        ELSE
            set next event of curr as next
        ENDIF
    } // End of case
} // End of switch
IF curr is not null THEN
    set last=curr
ELSE
    set next to null
ENDIF
ENDIF
ENDFOR

```

The traversal of the MSCFG identifies the number of scenarios that a sequence diagram has and the message-sequence of each scenario. The value of the 'Precondition' field of a message represents the message-sequence that has been sent before the current message. The first message of a scenario has no value for the 'Precondition' field. The value of 'Sequence-No' field of a message is assigned an integer number 'n' such that it represents the sequence of that message in the scenario.

The pseudo code for identifying the scenarios from the 'MSCFG' is given below:

```

Procedure      traverseMSCFG()
Input          Message-Sequence control Flow Graph
Output         scenarioList

Node   start;           /* start node of the graph */
String seq              /* current scenario */
String scenarioList     /* list of scenarios */

1.  set curr = start;
2.  IF curr is not null THEN
      add details represented by start 'seq'
      IF seq does not exist in scenarioList THEN
          add 'seq' to 'scenarioList'
      ENDIF
    ELSE
      IF curr is not conditional event THEN
          (i) add 'curr' to 'seq'
          (ii) recursively apply traverse procedure by setting the next event of curr as the new start
        ELSE
          (i) add 'curr' to 'seq'
          (ii) recursively apply traverse procedure by setting the true part of curr as the new start
          (iii) recursively apply traverse procedure by setting the false part of curr as the new start
        ENDIF
      ENDIF
    ENDIF

```

The 'Message-Sequences' extracted for scenario 1 and scenario 2 for the MSCFG (Figure 6.4) are shown in Table 6.7 and Table 6.8 respectively. In Table 6.7, the value for the 'Precondition' field of the message 'A5' is 'A1,A2,A3,A4,{(R1=TRUE)(T)}' where the term '{(R1=TRUE)(T)}' means that the result of the condition expression '(R1=TRUE)' evaluates as "True". Similarly, the value for the 'Precondition' field of the message 'A7' in Table 6.8 is 'A1,A2,A3,A4,{(R1=TRUE)(F)}' where the term '{(R1=TRUE)(F)}' means that the result of the condition expression '(R1=TRUE)' evaluates as "False".

Table 6.7: Message-Sequence For Scenario 1

Message#	Label	Precondition	Sequence-No
A1	1		1
A2	1.1	A1	2
A3	M2()	A1,A2	3
A4	M1()	A1,A2,A3	4
A5	2	A1,A2,A3,A4{(R1=TRUE)(T)}	5
A6	M3()	A1,A2,A3,A4{(R1=TRUE)(T)},A5	6

Table 6.8: Message-Sequence For Scenario 2

Message#	Label	Precondition	Sequence-No
A1	1		1
A2	1.1	A1	2
A3	M2()	A1,A2	3
A4	M1()	A1,A2,A3	4
A7	3	A1,A2,A3,A4{(R1=TRUE)(F)}	5
A8	M4()	A1,A2,A3,A4{(R1=TRUE)(F)},A7	6

The value of the ‘Sequence-No’ field of any message is assigned based upon its sequential order in a scenario and it is possible that a same ‘Sequence-No’ value is assigned to two different messages in two different scenarios. For example, in Table 6.7, the ‘Sequence-No’ value ‘6’ is assigned to the message ‘A6’ whereas the same ‘Sequence-No’ value is assigned to the message ‘A8’ in Table 6.8.

6.3.3 Step III: Identifying ‘Event-Sequence-State-Transitions’ for each Scenario

This step takes the state diagrams of the participating objects and ‘Message-Sequences’ of various scenarios as the inputs and identifies the ‘Event-Sequence-State-Transitions’ for each scenario.

The following conditions should be fulfilled before the execution of this step.

- (i) The state transition events of the state diagrams are to be indicated by the ‘Message#’ (message number) value of the message.
- (ii) If the state transition pattern of a participating object is different during the execution of a scenario, then a separate state diagram of that object for each scenario is drawn.

The pseudocode for this step is given below

The pseudo code for identifying all ‘Event-Sequence-State-Transitions’ is given below.

```

Procedure      identifyEvent-Sequence-State-TransitionTables()
Input          State Chart XMI file(s) and ‘Message-Sequences’ of Each Scenario
Output         Event-Sequence-State-Transition table corresponding to each Event-Sequence
                  table
/* Pseudo Code Description*/
1.FOR each ‘Message-Sequence’ extracted in Step I of the approach
        create an associated ‘Event- Sequence-State-Transition’;
      ENDFOR
2.FOR each Event-Sequence-State-Transition created
      (i) Use procedure populateEventSequence() of Chapter 5 to identify event sequence of
          the scenario
      (ii) FOR each component-name column of the Event-Sequence-State-Transition
           select the associated state diagram XMI file;
           read the initial_state for the component-name from XMI file;
           FOR each event number ‘E’ of the selected Event-Sequence-State-Transition
               scan the selected XMI file for state transition corresponding to ‘E’;
               IF ‘E’ is responsible for any state transition for the component
                   THEN
                       read next_state_transition for component-name from the
                           XMI file;
                       set the new value of initial_state as next_state_transition;
                       update component-name column with next_state_transition;
                   ELSE
                       update component-_name column with initial_state;
                   ENDIF
           ENDFOR
      ENDFOR
ENDFOR

```

The working of this step is in the following sections by taking the objects as shown in the sequence diagram of Figure 6.3(a) as an example.

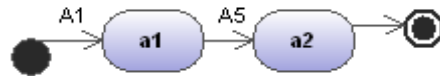
The arbitrary states of the participating objects ‘A’, ‘B’, ‘C’ and ‘D’ as {a1, a2, a3}, {b1, b2, b3}, {c1, c2, c3} and {d1, d2} respectively are used as examples. The initial states of the objects ‘A’, ‘B’, ‘C’ and ‘D’ are arbitrarily assumed as ‘a1’, ‘b1’, ‘c1’ and ‘d1’ respectively, and it is also arbitrarily assumed that the state transition pattern of object ‘A’ is different for two scenarios. The state transition pattern of other objects, namely ‘B’, ‘C’ and ‘D’ are assumed as same for both scenarios.

The state diagrams of object ‘A’ for scenario 1 and scenario 2 are shown in Figure 6.5(a) and Figure 6.5(b) respectively. Similarly, the state diagrams for objects ‘B’, ‘C’ and ‘D’

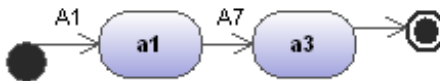
are drawn as shown in Figure 6.5(c), Figure 6.5(d) and Figure 6.5(e), respectively. It is to be noted that the state transition patterns for all the objects as shown in Figure 6.5 are assumed arbitrarily to demonstrate the working logic of this step.

The ‘Event-Sequence-State-Transitions’ identified for scenario 1 is shown in Figure 6.6(b). The ‘Message-Sequence’ for scenario 1 as shown in Table 6.7 and the state diagrams of objects ‘A’, ‘B’, ‘C’ and ‘D’ as shown in Figure 6.5(a), Figure 6.5(c), Figure 6.5(d) and Figure 6.5(e) are used as inputs in the identification of these ‘Event-Sequence-State-Transitions’.

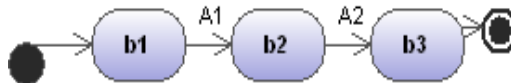
Similarly, the ‘Event-Sequence-State-Transitions’ identified for scenario 2 is shown in Figure 6.7(b). The ‘Message-Sequence’ for scenario 2 as shown in Table 6.8 and the state diagrams of objects ‘A’, ‘B’, ‘C’ and ‘D’ as shown in Figure 6.5(b), Figure 6.5(c), Figure 6.5(d) and Figure 6.5(e) are used as inputs in the identification of these ‘Event-Sequence-State-Transitions’.



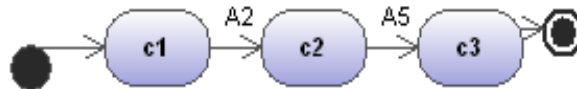
(a) State Diagram for Object A for Scenario I



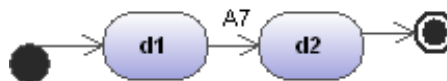
(b) State Diagram for Object A for Scenario II



(c) State Diagram for Object B



(d) State Diagram for Object C



(e) State Diagram for Object D

Figure 6.5: State Diagrams for the Participating Objects

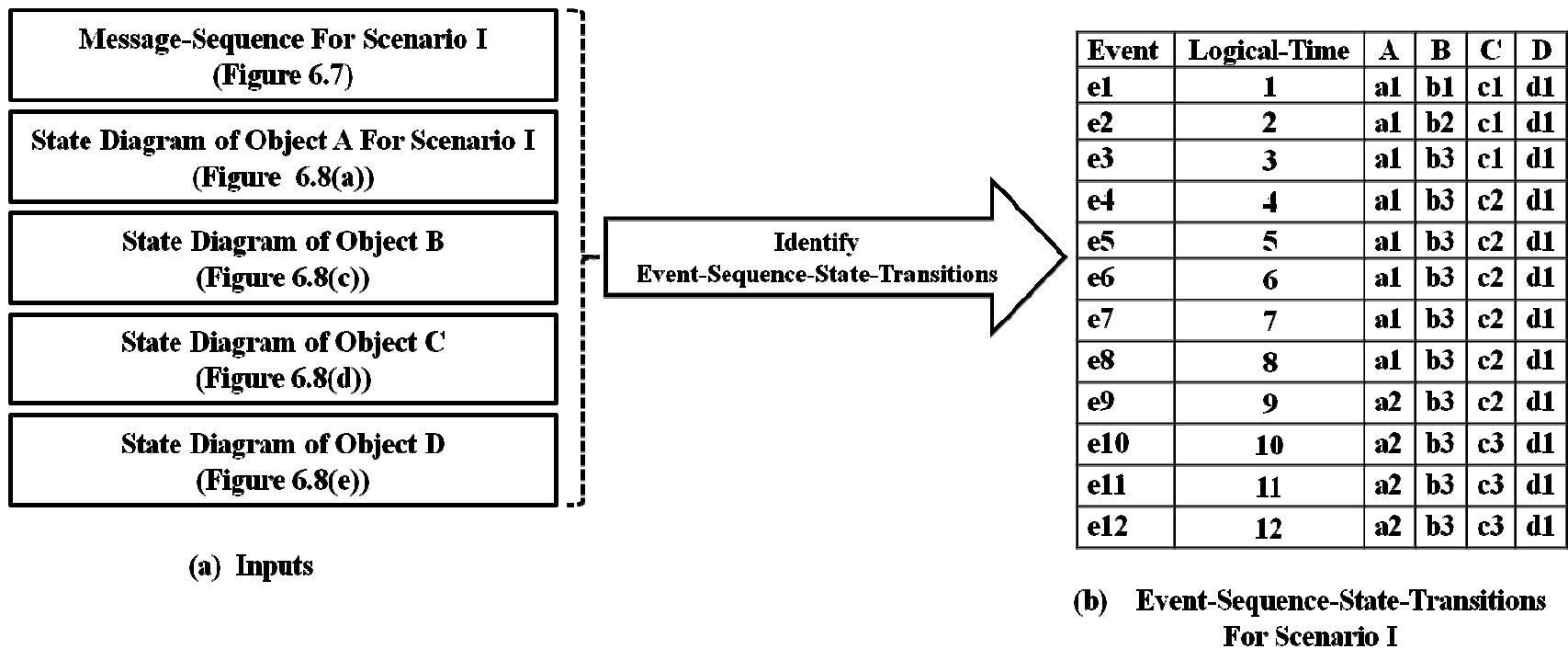


Figure 6.6: Execution of Step III of the Proposed Approach for Scenario 1

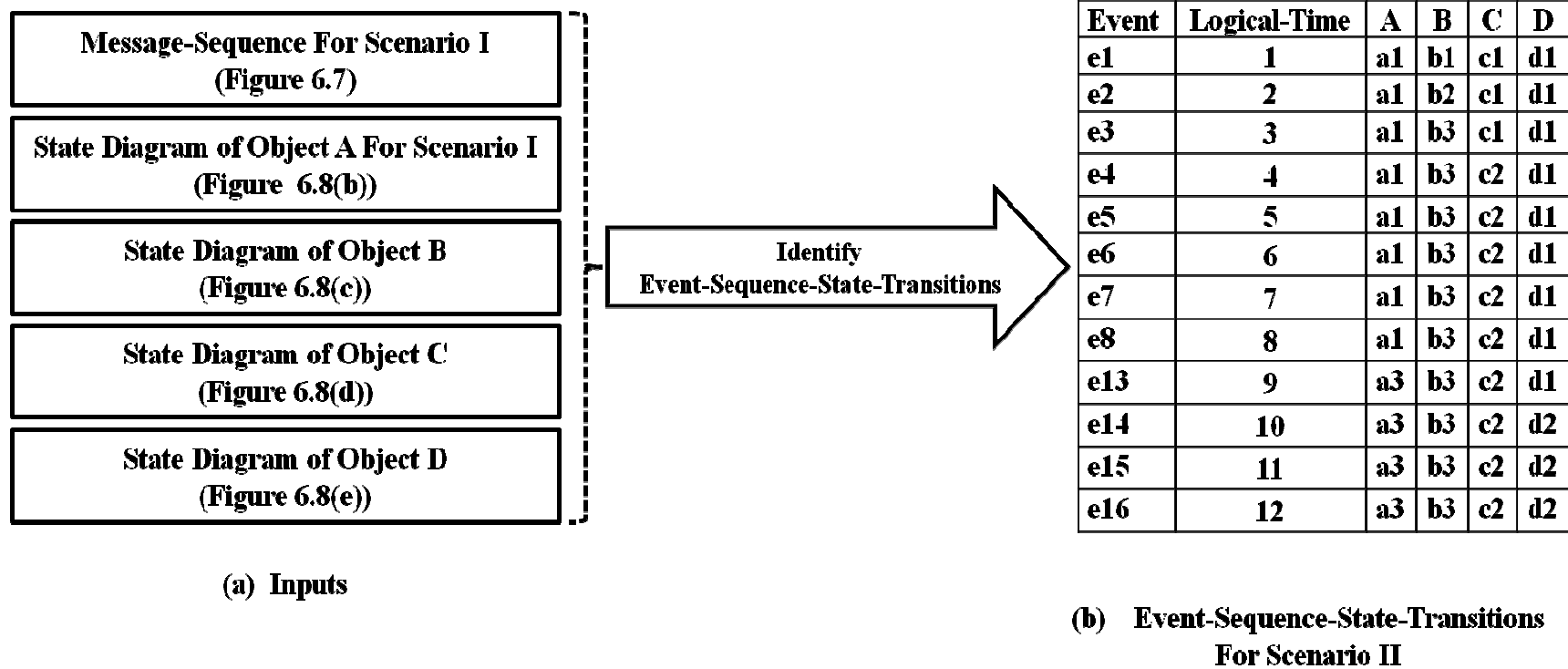


Figure 6.7: Execution of Step III of the Proposed Approach for Scenario 2

6.3.4 Perform Message-Errors-Effects-Analysis For Each ‘Event-Sequence-State-Transitions’

The objective of this step is to do the failure analysis of various messages-related errors for a scenario. Two tasks are carried out in this step, namely (i) Identification of Various Message-Related Errors and (ii) Investigating the effects of these errors on the system.

There can be any number of errors associated with a particular message. The messages-related errors that are only considered in the proposed approach and these are categorized into five types. The meaning and syntax of these types are shown in Table 6.9.

Table 6.9: Classification of Message-Related Error Categories

Message-Error Type	Purpose	Syntax	Example
Type I	This type of error represents the error situation where a sender object fails to send the required message. This type of error is only for send type of messages	‘!(M)’ where ‘M’ is a message-number and ‘!(M)’ means ‘M’ is not sent	!(A1) i.e. Message A1 is not sent
Type II	The sender object sends the message when its associated condition is not satisfied .This type of error is for those messages which either have a precondition explicitly assigned or have a condition in their computed precondition field	‘^(C) AND (M)’ where ‘M’ is a message-number and ‘C’ is the precondition for message ‘M’. ‘^(C)’ represents that C is wrongly evaluated as satisfied	^(door=closed) AND (A3) i.e. door is wrongly evaluated as closed and message A3 is sent.
Type III	The reply type, message carries ‘true’ response ‘whereas the response should be ‘false’. This type of error is only for reply type message whose response type is boolean.	‘(M)FT’ Where ‘M’ is a reply type message and it should carry a false value, but it is carrying a true value	(A4)FT
Type IV	The reply type, message carries ‘false’ response ‘whereas the response should be ‘true’. This type of error is also for reply type message only whose response type is boolean.	‘(M)TF’ Where ‘M’ is a reply type message and it should carry a true value, but it is carrying a false value	(A4)TF
Type V	The reply to message carries the incorrect result or computation ‘value’. This type of error is only for reply type message whose response type is not boolean.	(M)→ Where M is a reply message, which carries wrong numeric value.	(A6)’ i.e. A6 carries a wrong numeric value

The ‘Message-Errors-Effects-Analysis’ for a scenario has three main fields as shown in Table 6.4. The ‘Effects’ column is a matrix of size $n \times n$ where ‘ n ’ is the number of events in the corresponding ‘Event-Sequence-State-Transitions’ of a scenario.

The proposed SFMEA algorithm investigates the effects of various types of message-related errors as per the description given in the following subsections.

(i) Investigating The Effects of Type I Message-Related Errors

If an object fails to send a message, then it can be concluded reliably that all the messages that are in the ‘Message-Send-Dependency-List’ (‘Message-Details’ extracted in Step I(b)) of that message are also not sent. But, the messages that are in the ‘Message-Send-Independent-List’ (‘Message-Details’ extracted in Step I(b)) of the message may be sent. Therefore, a system can experience two types of state level error effects in this case. The first types of effects are felt because of not sending the messages and the second types of effects are observed because of wrong sending of the messages.

If any message is not sent then all the required state transitions that are occurring because of that message, will not occur (‘!’ sign is used to represent this effect such as $A \neq a$ which indicates that the required state transition in the state of the object ‘A’ has not occurred).

But if the messages in the ‘Message-Send-Independent-List’ of the message are sent by the object when its predecessor message is not sent during the same interaction by the same sender object, then it results in the wrong/erroneous state transition either for the same or for some other object. For example, the state level effect ‘ $A \neq a2$ ’ indicates the situation where object ‘A’ has wrongly or erroneously changed its state to ‘a2’. However, in this case the effects are conditional which means that the stated effects are observed only if the messages the ‘Message-Send-Independent-List’ of the message are sent. This fact is represented by inserting a message number at the end of the ‘Effects’ entry. For example, an ‘Effects’ entry such as ‘ $A \neq a2 (A5)$ ’ represents that the state of an object ‘A’ will be changed erroneously to ‘a2’ if the message ‘A5’ is sent.

(ii) Investigating the Effects of Type II Message-Related Errors

The effects of these types of errors are to be identified under the assumption that the receiver object of the message does not properly check for the precondition violation and erroneously starts the treatment of the message. It results in the wrong or erroneous state transitions, which are occurring after the send event of the message.

(iii) Investigating the Effects of Type III and Type IV Message-Related Errors

These types of errors are associated with the errors in the reply types of messages. The effects of these types of errors are observed in the following two ways.

- (a) Firstly, their effect is observed on the response value of the reply message that is sent immediately after the selected reply message. The value of the successive reply message is also erroneously changed.
- (b) Secondly, these errors affect the result of a conditional decision, especially when the same reply message is used in some conditional evaluation. It results in the wrong execution of the scenario, which further results in the wrong/erroneous state transitions for the objects whose states are changed in the executed scenario.

It should be noted that the effects of these types of errors are scenario specific and it is possible that a selected error of this type is shown as having no effect in one scenario, but the same error may be shown as having some state level effect in some other scenario.

(iv) Investigating the effects of Type V Message-Related Errors

These types of errors are associated with reply messages that return the result of some computation. In this case, it is assumed that the value of the result that a reply message carries is wrong. The effects of this type of error are felt in three ways. The first two types of effects are same as discussed in the case (iii) discussed above. The third type of impact is felt when the reply message is used as a parameter for some other send type message. If a wrong value has been sent as a parameter, then it results in the erroneous state transitions in the states of the objects that are affected by the message.

The pseudo code of the algorithm for this step is given below:

```

Procedure      Perform-Message-Errors-Effects-Analysis()
Input(s)      Output(s) of Step I, Step II and Step III
Output(s)     Message-Errors-Effects-Analysis For Each Scenario

1.  Create a Message-Error-Effects-Analysis for each Event-Sequence-State-Transitions with Message#
2.  FOR each Message-Error-Effects-Analysis table
      /* Define Message Errors for various messages */
      FOR each message 'm' in the Message-Error-Effects-Analysis table
        Identify the message errors corresponding to message 'm';
        /* Finding out the Effects of Errors */
        FOR each message error 'msgErr' identified
          Case: 'msgErr' is Type I

```

```

{
  Case: message 'm' is a send type message with a reply message 'r'
  (a) Find the objects whose state are changed between the
      send event 'x' of message 'm' and the receive event 'y' of
      reply message 'r'
  (b) Mark the states for all the objects as not not changed, i.e.
      those objects will not be able to change their respective
      states
  (c) FOR each message in the send-independent-list of
      message 'm'
      (i) Find out the objects whose state are changed during
          the sending of the message
      (ii) Mark the states for each such object as erroneously
          changed
      ENDFOR
  Case : message 'm' is a send type, message without a reply message
  (a) FOR each message in the dependency-list of message 'm'
      (i) Find out the objects whose state are changed during
          the sending of the message
      (ii) Mark the states for each such object as not changed
      ENDFOR
  (b) FOR each message in the send-independent-list of
      message 'm'
      (i) Find out the objects whose state are changed during
          the sending of the message
      (ii) Mark the states for each such object as erroneously
          changed
      ENDFOR
} // End of Type I

```

Case : 'msgErr' is Type II

```

{
  (i) Identify the state transitions that are occurring after the send
      event of the message 'm'
  (ii) Mark all states as erroneous state transitions using the symbol
      '^='
} // End of Type II

```

Case : 'msgErr' is Type III or IV or V

```

{
  (i) Identify the reply message (if any) that is sent after this reply
      message.
  (ii) Record its effect corresponding to the send event of that reply
      message.
  (iii) IF message is associated with conditional evaluation THEN
      (a) Identify the scenario that is affected by the error.
}

```

- (b) *Identify the objects whose states are changed after the send of this reply message*
- (c) *Mark the states of all those objects as changed erroneously using symbol '^='*

ENDIF

}// End of Type III

ENDFOR

ENDFOR

ENDFOR

The 'Message-Errors-Effects-Analysis' results for scenario 1 is shown in Table 6.10.

The 'Effects' column of the 'Message-Errors-Effects-Analysis' for scenario 1 as shown in Table 6.10 is divided into 12 event sub-columns with titles as 'e1', 'e2', 'e3', 'e4', 'e5', 'e6', 'e7', 'e8', 'e9', 'e10', 'e11' and 'e12'. These events are selected from the 'Event-Sequence-State-Transitions' for scenario 1 as shown in Figure 6.6(b).

The 'Message-Errors-Effects-Analysis' of scenario 2 is shown in Table 6.11. The 'Effects' column of the 'Message-Errors-Effects-Analysis' for scenario 2 as shown in Table 6.11 is divided into 12 event sub-columns with titles as 'e1', 'e2', 'e3', 'e4', 'e5', 'e6', 'e7', 'e8', 'e13', 'e14', 'e15' and 'e16'. These events are selected from the 'Event-Sequence-State-Transitions' for scenario 2 as shown in Figure 6.7(b).

The message-related errors are identified by taking 'Message-Details' as shown in Table 6.6 as an input. The message number (Message#) 'A1' (Table 6.6) is a send type message so only one message-error as '!(A1)' is identified for it in Table 6.10. Similarly, one error '!(A2)' is defined for message 'A2' in Table 6.10 for a similar reason. The messages 'A3' and 'A4' in Table 6.6 are a reply type messages with a 'boolean' response type and that's why two message-errors for messages 'A3' and 'A4' namely {'(A3)FT', '(A3)TF'} and {'(A4)FT', '(A4)TF'} respectively, are identified in Table 6.10. The message 'A6' in Table 6.6 is reply type message with a 'value' type response and that is why only one message-error namely '(A6)' is identified for it in Table 6.10 only because the message 'A6' only appears in scenario 1.

It is to be noted that if a message appears in the 'Message-Sequence' of both scenarios, then the message-errors associated with that message also appear in the 'Message-Errors-Effects-Analysis' of both scenarios.

Table 6.10: Message-Errors-Effects-Analysis for Scenario 1

Message#	Message-Errors	Effects											
		e1	e2	e3	e4	e5	e6	e7	e8	e9	e10	e11	e12
A1	!(A1)	A!=a1	B!=b2	B!=b3	C!=c2					A^=a2(A5)	C^=c3(A5)		
A2	!(A2)			B!=b3	C!=c2								
A3	(A3)FT							(A4)FT	(A4)FT	A^=a2	C^=c3		
	(A3)TF							(A4)TF	(A4)TF				
A4	(A4)FT												
	(A4)TF									A^=a2	C^=c3		
A5	!(A5)									A!=a2	C!=c3		
A6	(A6)'											(A6)'	(A6)'

Table 6.11: Message-Errors-Effects-Analysis for Scenario 2

Message#	Message-Errors	Effects											
		e1	e2	e3	e4	e5	e6	e7	e8	e13	e14	e15	e16
A1	!(A1)	A!=a1	B!=b2	B!=b3	C!=c2					A^=a3(A7)	D^=d2(A7)		
A2	!(A2)			B!=b3	C!=c2								
A3	(A3)FT							(A4)FT	(A4)FT	A^=a3	D^=d2		
	(A3)TF							(A4)FT	(A4)FT				
A4	(A4)FT												
	(A4)TF									A^=a3	D^=d2		
A7	!(A7)									A!=a3	D!=d2		
A8	(A8)'											(A8)'	(A8)'

For example, the message-errors ‘!(A1)’ and ‘!(A2)’ appears in both Table 6.10 and Table 6.11 because the messages ‘A1’ and ‘A2’ appear in the ‘Message-Sequences’ of both the scenarios as shown in Table 6.7 and Table 6.8. On the other hand, the message-error ‘(A6)’ appears only in Table 6.10 because the message ‘A6’ appears only in the ‘Message-Sequence’ of scenario 1 (Table 6.7). Similarly, the message-error ‘(A8)’ appears only in Table 6.11 because the message ‘A8’ appears only in the ‘Message-Sequence’ of scenario 2 (Table 6.8).

The ‘Message-Details’ Table 6.6 and ‘Message-Errors-Effects-Analysis’ for scenario 1 and scenario 2 in Table 6.10 and Table 6.11 are used to explain the effects of various types of message-related errors in the following sections.

(a) Investigating the Effects of the Message Error !(A1) i.e. message A1 is not sent

If message ‘A1’ is not sent (i.e.!(A1)) then the message ‘A2’ will not be sent too, because the message ‘A2’ is in the ‘Message-Send-Dependency-List’ of the message ‘A1’ (see Table 6.6). The message ‘A1’ has an associated reply message ‘A4’ with message name as ‘R1(boolean)’ (Table 6.6). So, all the state transitions that are occurring between the send-event of message ‘A1’ (i.e. event ‘e1’ in Table 6.6) and the receive-event of message ‘A4’ (i.e. event ‘e8’ in Table 6.6) will not occur. As per the ‘Event-Sequence-State-Transitions’ of scenario 1 as shown in Figure 6.6(b), the following state transitions are occurring between events ‘e1’ and ‘e8’.

- (i) The state of an object ‘A’ is changed to state ‘a1’ during event e1
- (ii) The state of an object ‘B’ is changed to state ‘b2’ during event e2
- (iii) The state of an object ‘B’ is changed to state ‘b3’ during event e3
- (iv) The state of an object ‘C’ is changed to state ‘c2’ during event e4

Therefore, all the above-mentioned state transitions will not occur in the system if the message ‘A1’ is not sent. These effects are shown in Table 6.10 as ‘A!=a1’, ‘B!=b2’, ‘B!=b3’ and ‘C!=c2’ under the respective event columns, e1, e2, e3 and e4.

The ‘Message-Send-Independent-List’ of the message ‘A1’ in Table 6.6 has two messages as ‘A5’ (with ‘Label’ value as 2) and ‘A7’ (with ‘Label value as 3). If the message ‘A5’ is sent then its effect is observed in scenario 1 only because the message ‘A5’ is not included in scenario 2 (Table 6.8). The events of the message ‘A5’ result in the state transitions of two objects, namely ‘A’ and ‘C’ (Figure 6.6(b)). The object ‘A’ is

changing its state to 'a2' at event 'e9' and the object 'C' is changing its state to state 'c3' at event 'e10'(Figure 6.6(b)). These transitions are known as erroneous state transitions and are indicated as ' $A^{\wedge}=a2(A5)$ ' and ' $C^{\wedge}=c3(A5)$ ' under the respective event columns (events 'e9' and 'e10') in Table 6.10. It indicates that the state of the object 'A' is erroneously changed to 'a2' if the message 'A5' is sent when the message 'A1' is not sent.

The sending of the message 'A7' affects only the events of scenario 2 (because the message 'A7' is not included in scenario 1). The sending of the message 'A7' changes the state of the 'A' object to 'a3' and the state of the 'D' object to 'd2'. These effects are indicated ' $A^{\wedge}=a3(A7)$ ' and ' $D^{\wedge}=d2(A7)$ ' and are shown in Table 6.11 under the respective event columns (event 'e13' and 'e14').

(b) Investigating the Effects of the Message Error !(A2) i.e. message A2 is not sent

If the message 'A2' is not sent then its effects are observed in events 'e3' (send event of 'A2) and 'e4' (receive event of 'A2') only because there is no message in the 'Message-Send-Dependency-List' and the 'Message-Send-Independent-List' of the message 'A2'(Table 6.6). The effects of this error are same in both the scenarios and are shown in Table 6.10 and Table 6.11 respectively.

(c) Investigating the Effects of the Message Errors associated with the message 'A3'

The message 'A3' is a reply type message with a boolean (True/False) type response. Two types of message-related errors are associated with reply messages as shown in Table 6.9 and these are explained in following two sub-sections.

(d) Investigating the Effects of the '(A3)FT'

The first message-related error for message 'A3' is '(A3) FT'. It means that the message 'A3' should return a 'False' value, but it carries a 'True' value. This error impacts the following:

- The value of the reply message which is sent immediately after the current reply message
- Execution of the scenarios if the current reply message is used as a conditional expression

So, the first of this error is observed in the value of the response of the message 'A4'because it is sent immediately after 'A3'. The value of 'A4' is also changed to True.

The effect '(A4)FT' is shown under events e7 and e8 of Table 6.10 and table 6.11. The wrong response value of 'A3' is also transmitted in 'A4'.

The changed value of 'A4' results in the execution of scenario 1 because 'A4' is used in the conditional evaluation(Figure 6.2). [Note that the actual name of 'A4' is R1 (boolean) in Table 6.6]. So, all the state transitions occurring in scenario 1 after the message 'A3' are marked as erroneous state transitions (under events 'e9' and 'e10' in Table 6.10). The effects of this error are not observed in scenario 2.

(i) Investigating the Effects of the '(A3)TF'

The second message-related error in the message 'A3' is '(A3)TF'. It means that the message 'A3' should return a 'True' value, but it carries a 'False' value. The effects of this error are observed in scenario 2 because it results in the execution of scenario 2.

6.3.5 Time complexity of the Algorithm

(a) Time Complexity of Step I

The running time, i.e. the algorithmic time complexity of the Step I(a) is of the order of ' $O(N1)$ ' where ' $N1$ ' is the number of messages in the sequence diagram.

The running time, i.e. the algorithmic time complexity of the Step I(b) is also of the order of ' $O(N1)$ ' where ' $N1$ ' is the number of messages in the sequence diagram.

So the overall algorithmic time complexity of Step I is of the order of ' $O(N1)$ '.

(b) Time Complexity of Step II

The algorithmic time complexity of the Step II (a) is of the order of ' $O(N1)$ ' where ' $N1$ ' is the number of messages in the sequence diagram. The algorithmic time complexity of Step II (b) is of the order of ' $O(N2)$ ' where ' $N2$ ' is the number of nodes in the MSCFG because the algorithm is recursively applied to each node.

So the overall algorithmic time complexity of Step I is of the order of ' $O(N1)+O(N2)$ '.

(c) Time Complexity of Step III

The running time, i.e. the algorithmic time complexity of the Step I(a) is of the order of ' $O(N3 \times N4 \times N5)$ ' where ' $N3$ ' is the number of 'Event-Sequence-State-Transitions', ' $N4$ ' is the number of components for which a state diagram is drawn and ' $N5$ ' is the average number of events in each 'Event-Sequence-State-Transitions'.

(d) Time Complexity of Step IV

The running time of Step IV is of the order of ‘ $O(N3 \times N6 \times N7)$ ’ where ‘ $N3$ ’ is the number of ‘Event-Sequence-State-Transitions’, ‘ $N6$ ’ is the average number of messages in each scenario and ‘ $N7$ ’ is the average number of message-related errors corresponding to each message of the scenario.

Therefore, the whole algorithmic time complexity of all the four steps of SFMEA algorithm is given below:

$$[O(N1)] + [O(N1)+O(N2)] + [O(N3 \times N4 \times N5)] + [O(N3 \times N6 \times N7)]$$

6.3.6 Sequence and State Diagram Representations

In the SFMEA algorithm discussed above, it is assumed that the sequence and state diagrams are supplied in some specific representation form as follows:

- (i) The sequence diagram is drawn using nested message numbering as discussed in Section 6.3.
- (ii) Any send type message (synchronous or asynchronous) in the sequence diagram should have the following form

$$messageName(parameterList)\{messageLabel\}$$

where

- (a) *MessageName* represents the name of the message,
 - (b) *parameterList* represents the list of parameters that are passed along with the message. The *parameterList* has the form

$$(param1:type, param2:type, \dots, para-n:type)$$
 - (c) *messageLabel* is the message number assigned by the Altova UML tool and *it is mandatory to embed this number in the messageName*.
- (iii) Reply type of message in the sequence diagram should have the form: *replyMessage-Name(type)*. Where ‘*type*’ indicates the type of response carried by the reply message. Two types of responses are considered in the approach for a reply type message and these are : *boolean (True/False)* or *value (any type of numeric value)*.
 - (iv) The state diagrams should be drawn using the unique message numbers (*Message#*) assigned to various messages as state transition events.

6.4 APPLICATION OF SFMEA ALGORITHM TO SAFETY-CRITICAL SOFTWARE SYSTEMS

The proposed algorithm is applied on two safety-critical applications, namely Rail Track Door Control System (RTCS) and Insulin Delivery System (IDS) as discussed in Chapter 4. The step-by-step application of the algorithm on the sequence and state diagrams of these systems is given in the following sub-sections.

6.4.1 Motivating Example I: Railway Track Door Control System (RTCS)

This case study has been selected from the works of Medikonda and Swarup (Medikonda and Swarup, 2011) to demonstrate the application of the presented SFMEA approach. This system is used to automatically close the rail track door in case of arrival of the train. The message sequence diagram for the RTCS study is shown in Figure 6.8.

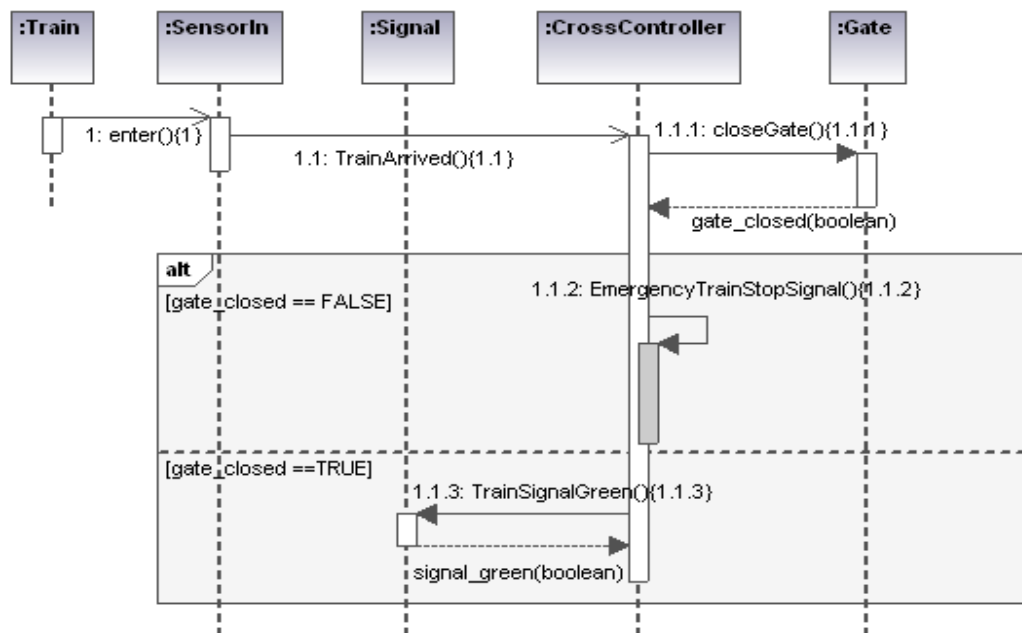


Figure 6.8: Message Sequence Diagram For Rail Track Door Controller System

There are five objects, namely ‘:Train’, ‘:SensorIn’, ‘:Signal’, ‘:CrossController’, and ‘:Gate’ that are participating in this use-case functionality. The functionality gets activated when the ‘:SensorIn’ object detects the arrival of a train and generates an interrupt for the ‘:CrossController’ object. Upon receiving of this interrupt message, the ‘:CrossController’ object first issues a command to close the door. When the door is closed, then it issues the turn green signal command. If a door failure is detected during operation, then an emergency train stop signal is issued by the ‘:CrossController’ object.

Step I: Generate Pseudo Code and Extract Message-Details

The pseudo code generated by applying Step I(a) to an XMI file of the sequence diagram of Figure 6.8 is shown in Figure 6.9 along with the numbering of messages and the ‘Message-Details’ extracted from the same input file by applying Step I(b) is shown in Table 6.12.

```

enter(){1}
TrainArrived(){1.1}
closeGate(){1.1.1}
gate_closed(Boolean)
IF gate_closed == FALSE THEN
EmergencyTrainStopSignal(){1.1.3}
ELSE
TrainSignalGreen(){1.1.3}
signal_green(Boolean)
ENDIF

```

Figure 6.9: Pseudo Code Form of the Sequence Diagram of Figure 6.13

The messages ‘M1’ and ‘M2’ are of asynchronous type and that is why their ‘Type’ value is 2. All other messages are of synchronous type and their ‘Type’ value is 1. The ‘Label’ value of the message ‘M1’ is ‘1’ and it is contained inside the ‘Label’ values of the messages ‘M2’ (with ‘Label’ value ‘1.1’), ‘M3’ (with ‘Label’ value ‘1.1.1’), M5 (with ‘Label’ value ‘1.1.2’) and ‘M7’ (with ‘Label’ value ‘1.1.3’). Therefore, the ‘Message-Send-Dependency-List’ of ‘M1’ contains the ‘Label’ values {1.1, 1.1.1, 1.1.2, 1.1.3}. Since, no message is sent after the message ‘M1’ by its sender object (‘:Train’), the ‘Message-Send-Independent-List’ of the message ‘M1’ contains no message.

Similarly, The ‘Label’ value of the message ‘M2’ is ‘1.1’ and it is contained inside the ‘Label’ values of the ‘M3’ (with ‘Label’ value ‘1.1.1’), M5 (with ‘Label’ value ‘1.1.2’) and ‘M7’ (with ‘Label’ value ‘1.1.3’). So, the ‘Message-Send-Dependency-List’ of ‘M2’ contains the ‘Label’ values {1.1.1, 1.1.2, 1.1.3}. Since, no message is sent after the message ‘M2’ by its sender object (‘:SensorIn’), the ‘Message-Send-Independent-List’ of the message ‘M2’ contains no message.

The ‘Label’ value of the message ‘M3’ is ‘1.1.1’ and it is contained in no other message’s ‘Label’ value. That’s why, the ‘Message-Send-Dependency-List’ of ‘M3’ contains no message. But there are two messages, namely ‘M5’ and ‘M7’ are sent after the message ‘M3’ by the sender object of ‘M3’ (‘: CrossController’). So the ‘Message-Send-Independent-List’ of the message ‘M3’ contains the labels of message ‘M5’ (‘Label’ value ‘1.1.2’) and ‘M7’ (‘Label’ value ‘1.1.3’).

Table 6.12 : Message-Details Created For Rail Track Door Controller Application

Message#	Message-Name	Label	From	To	Type	isReply	Reply-Message	Send Event	Receive Event	Message-Send-Dependency-List	Message-Send-Independent-List
M1	enter()	1	Train	SensorIn	2	0		e1	e2	1.1, 1.1.1, 1.1.2, 1.1.3	
M2	TrainArrived()	1.1	SensorIn	CrossController	2	0		e3	e4	1.1.1, 1.1.2, 1.1.3	
M3	closeGate()	1.1.1	CrossController	Gate	1	0	gate_closed(boolean)	e5	e6		1.1.2, 1.1.3
M4	gate_closed(boolean)	closeGate()	Gate	CrossController	1	1		e7	e8		
M5	EmergencyTrainStopSignal()	1.1.2	CrossController	CrossController	1	0		e9	e10		
M6	TrainSignalgreen()	1.1.3	CrossController	signal	1	0	signal_green(boolean)	e11	e12		
M7	Signal_green(boolean)	TrainSignalgreen()	Signal	CrossController	1	1		e13	e14		

Step II: Extract Message-Sequence for Each Scenario

The sequence diagram as shown in Figure 6.8 has two scenarios because of an ‘alt’ block involving the condition ‘gate-closed = FALSE’. The ‘Message-Sequence’ of the scenario 1 and scenario 2 are shown in Table 6.13 and Table 6.14 respectively.

Table 6.13: Message-Sequence for Scenario 1 of RTCS Application

Message#	Label	Precondition	Sequence-No
M1	1		1
M2	1.1	M1	2
M3	1.1.1	M1,M2	3
M4	closeGate()	M1,M2,M3	4
M5	1.1.2	M1,M2,M3,M4,(gate_closed=FALSE)(T)	5

Table 6.14: Message-Sequence for Scenario 2 of RTCS Application

Message#	Label	Precondition	Sequence-No
M1	1		1
M2	1.1	M1	2
M3	1.1.1	M1,M2	3
M4	closeGate()	M1,M2,M3	4
M6	1.1.2	M1,M2,M3,M4,(gate_closed=FALSE)(F)	5
M7	signal_green (boolean)	M1,M2,M3,M4,(gate_closed=FALSE)(F),M6	6

Step III : Identify ‘Event-Sequence-State-Transitions’ For Each Scenario

The state diagrams drawn for three objects, namely ‘: CrossController’, ‘:Gate’ and ‘:Signal’ are shown in Figure 6.10.



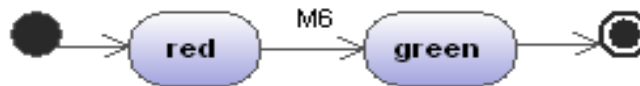
(a) CrossController State Diagram for Normal Scenario



(b) CrossController State Diagram for Emergency Scenario



(c) Gate State Diagram



(d) Signal State diagram

Figure 6.10: State Diagrams for Rail Track Door Controller Application

Two state diagrams are drawn for the ‘:CrossController’ object (one for normal scenario and one for emergency scenario) whereas only one state diagram is drawn for the ‘:Signal’ and ‘:Gate’ objects. The initial states of the ‘: CrossController’, ‘:Gate’ and ‘:Signal’ objects are assumed as ‘waiting’, ‘open’ and ‘red’ respectively.

The ‘Event-Sequence-State-Transitions’ for scenario 1 and scenario 2 generated by the application of Step II are shown in Table 6.15 and Table 6.16 respectively.

Table 6.15: ‘Event-Sequence-State-Transitions’ for Scenario 1 of RTCS Application

Event	Logical-Time	:CrossController	:Gate	:Signal
e1	1	waiting	open	red
e2	2	waiting	open	red
e3	3	waiting	open	red
e4	4	servicing	open	red
e5	5	gate_closing	open	red
e6	6	gate_closing	open	red
e7	7	gate_closing	open	red
e8	8	gate_closing	open	red
e9	9	emergency_trainstopping	open	red
e10	10	emergency_trainstopping	open	red

Table 6.16: ‘Event-Sequence-State-Transitions’ for Scenario 2 of RTCS Application

Event	Logical-Time	:CrossController	:Gate	:Signal
e1	1	waiting	open	red
e2	2	waiting	open	red
e3	3	waiting	open	red
e4	4	servicing	open	red
e5	5	gate_closing	open	red
e6	6	gate_closing	closed	red
e7	7	gate_closing	closed	red
e8	8	gate_closing	closed	red
e11	9	green_signaling	closed	red
e12	10	green_signaling	closed	green
e13	11	green_signaling	closed	green
e14	12	green_signaling	closed	green

Step IV: Perform ‘Message-Errors-Effects-Analysis’ of Each Scenario

The ‘Message-Errors-Effects-Analysis’ for scenario 1 and scenario 2 is performed by applying Step IV and the results are shown in Table 6.17 and Table 6.18 respectively.

Table 6.17: Message-Errors-Effects-Analysis for Scenario 1 of RTCS Application

Message#	Message-Errors	Effects									
		e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
M1	!(M1)				crossController !=servicing	crossController !=gate_closing	gate!=closed			crossController !=emergency_trainstopping	
M2	!(M2)				crossController !=servicing	crossController !=gate_closing	gate!=closed			crossController !=emergency_trainstopping	
M3	!(M3)					crossController !=gate_closing	gate!=closed			crossController ^=emergency_trainstopping (M5)	
M4	(M4)FT										
	(M4)TF									crossController ^=emergency_trainstopping (M5)	
M5	!(M5)									crossController !=emergency_trainstopping	

Table 6.18: Message-Errors-Effects-Analysis for Scenario 2 of RTCS Application

Message#	Message-Errors	Effects											
		e1	e2	e3	e4	e5	e6	e7	e8	e11	e12	e13	e14
M1	!(M1)				crossController !=servicing	crossController !=gate_closing	gate!=closed			crossController !=green_signalling	signal!=green		
M2	!(M2)				crossController !=servicing	crossController !=gate_closing	gate!=closed			crossController !=green_signalling	signal!=green		
M3	!(M3)					crossController !=gate_closing	gate!=closed			crossController ^=green_signalling (M6)	signal^=green (M6)		
M4	(M4)FT									crossController ^=green_signalling (M6)			
	(M4)TF												
M6	!(M6)									crossController !=green_signalling	signal!=green		
M7	(M7)TF												
	(M7)FT												

6.4.2 Motivation Example II: Insulin Delivery System (IDS)

This case study has been discussed in Chapter 4 while applying the SFMEA approach in use-case based requirements analysis phase. The objects required to implement the ‘Deliver Insulin’ use-case functionality are shown in the sequence diagram of Figure 6.11. There are six objects participating in this functionality and these are ‘:Clock’, ‘:Controller’, ‘:Sensor’, ‘:InsulinCompute’, ‘:InsulinPump’ and ‘:Display’.

The whole interaction starts when an interrupt message ‘changeState (RUN)’ is received by the ‘: Controller’ object from the ‘:Clock’ object. Upon the receipt of this interrupt message, the following actions are carried out by the ‘:Controller’ object in sequence.

- (i) The ‘:Controller’ first measures the current sugar level in the body
- (ii) If the sugar level is in the acceptable range, then the functionality gets exited and no insulin is injected in the body.
- (iii) If the sugar level is high, then the ‘: controller’ computes the amount of insulin to be injected so as to bring the sugar level under control.
- (iv) After computing the value for the required amount of insulin, the ‘:Controller’ object instructs the insulin pump to inject the computed amount of insulin in the patient’s body.
- (v) If the sugar level is within acceptable limits, then the ‘: controller’ flashes the suitable message on the display of the system

The role of the ‘: InsulinCompute’ class is to compute the amount of insulin required to be injected into the patient’s body. The role of the ‘:InsulinPump’ class is to inject the requested amount of insulin in the body. The ‘:Display’ class is required to flash the failure message on the display when insulin is not injected in the body.

Step I: Generate Pseudo Code and Extract Message-Details

The pseudo code form generated for the sequence diagram of Figure 6.11 is shown in Figure 6.12.

The ‘Message-Details’ generated for this application is shown in Table 6.19. The messages ‘M1’, ‘M8’ and ‘M9’ are of asynchronous type and that’s why their ‘Type’ field value is ‘2’. All other messages are of synchronous type and that’s why their ‘Type’ field value is ‘1’.

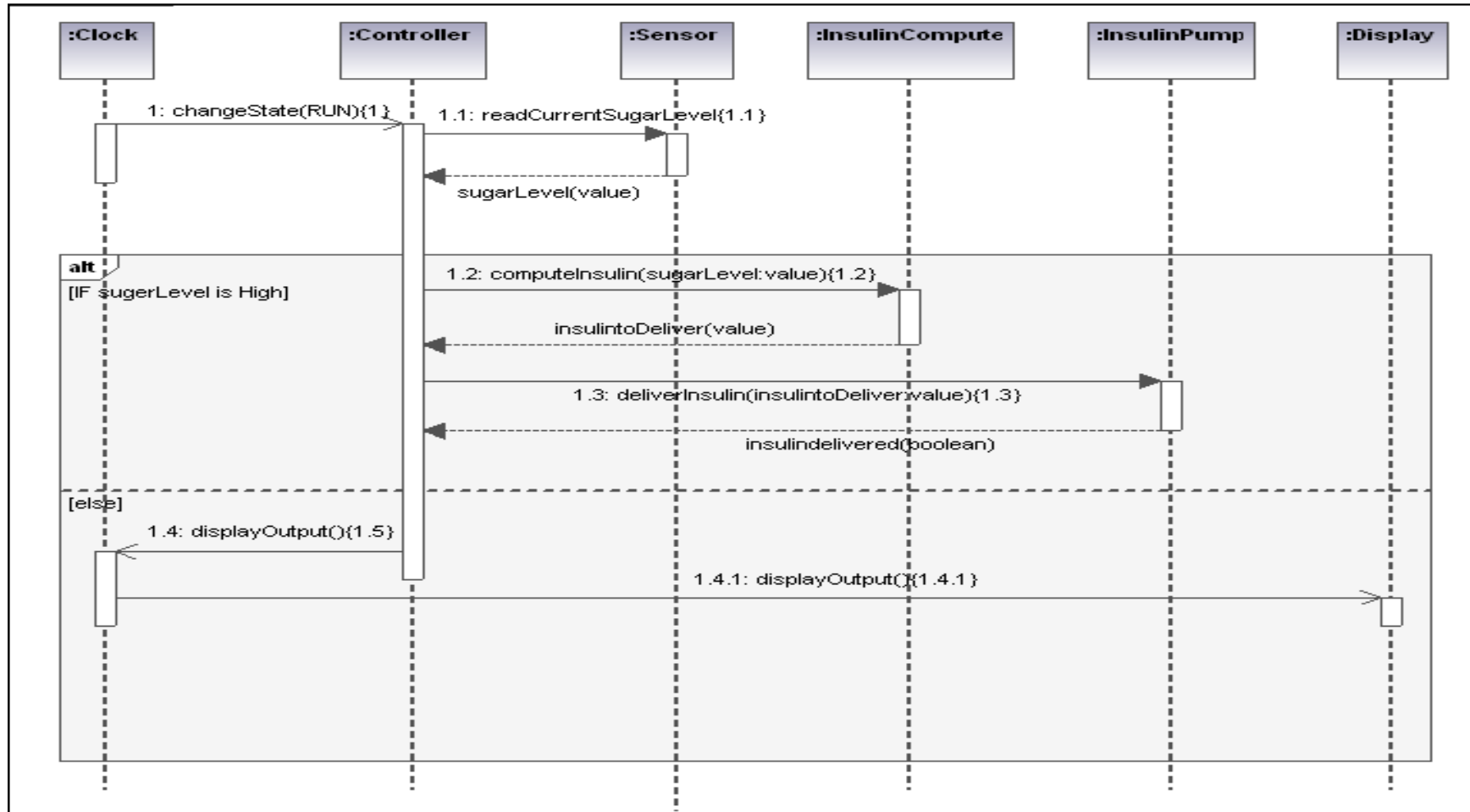


Figure 6.11: Sequence Diagram for Insulin Delivery Pump System

Table 6.19: Message-Details Extracted For Insulin Delivery System

Message#	Message-Name	Label	From	To	Type	isReply	Reply-Message	Send Event	Receive Event	Message-Send-Dependency-List	Message-Send-Independent-List
M1	changeState(RUN)	1	Clock	Controller	2	0		e1	e2	1.1, 1.2, 1.3, 1.4, 1.4.1	
M2	readCurrentSugarLevel	1.1	Controller	Sensor	2	0	sugarLevel(value)	e3	e4		1.2, 1.3, 1.4
M3	sugarLevel(value)	readCurrentSugarLevel	Sensor	Controller	1	1		e5	e6		
M4	computeInsulin(sugarLevel:value)	1.2	Controller	InsulinCompute	1	0	insulintoDeliver(value)	e7	e8		1.3, 1.4
M5	insulintoDeliver(value)	computeInsulin(sugarLevel:value)	InsulinCompute	Controller	1	1		e9	e10		
M6	deliverInsulin(insulintoDeliver:value)	1.3	Controller	InsulinPump	1	0	insulindelivered(boolean)	e11	e12		1.4
M7	Insulindelivered(boolean)	deliverInsulin(insulintoDeliver:value)	InsulinPump	Controller	1	1		e13	e14		
M8	displayOutput()	1.4	Controller	Clock	2	0		e15	e16	1.4.1	
M9	displayOutput()	1.4.1	Clock	Display	2	0		e17	e18		

```

changeState(RUN)
readCurrentSugarLevel(){1,1}
sugarLevel(value)
IF sugar level is High THEN
computeInsulin(sugarLevel:value){1.2}
insulintodeliver(value)
deliverInsulin(insulintoDeliver:value){1.1}
insulindelivered(Boolean)
ELSE
displayOutput(){1.4}
displayOutput(){1.4.1}
ENDIF

```

Figure 6.12: Pseudo Code Form of the Sequence Diagram of Figure 6.13

Step II: Extract Message-Sequence For Each Scenario

The pseudo code description as shown in Figure 6.12 has two scenarios because of the presence of an ‘IF’ statement. The ‘Message-Sequence’ generated for scenario 1 and scenario 2 of this application are shown in Table 6.20 and Table 6.21 respectively.

Table 6.20: Message-Sequence For Scenario 1 of IDS Application

Message#	Label	Precondition	Sequence-No
M1	1		1
M2	1.1	M1	2
M3	readCurrentsugarLevel	M1,M2	3
M4	1.2	M1,M2,M3,(sugarLevel is High)(T)	4
M5	computeInsulin(sugar Level:value)	M1,M2,M3,(sugarLevel is High)(T),M4	5
M6	1.3	M1,M2,M3,(sugarLevel is High)(T),M4,M5	6
M7	deliverInsulin(insulintoDeliver:value)	M1,M2,M3,(sugarLevel is High)(T),M4,M5,M6	7

Table 6.21: Message-Sequence For Scenario 2 of IDS Application

Message#	Label	Precondition	Sequence-No
M1	1		1
M2	1.1	M1	2
M3	readCurrentsugarLevel	M1,M2	3
M8	1.4	M1,M2,M3,(sugarLevel is High)(F)	4
M9	1.4.1	M1,M2,M3,(sugarLevel is High)(F),M8	5

Step III: Identify ‘Event-Sequence-State-Transitions’ For Each Scenario

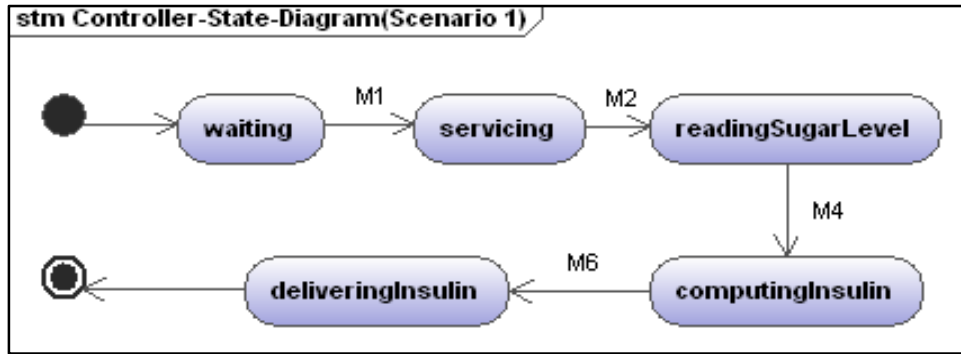
The state diagrams are supplied for three objects in this application and these are (i) ‘:Controller’, (ii) ‘:Sensor’ and (iii) ‘:InsulinPump’. The initial states of the ‘:Controller’, ‘:Sensor’ and ‘:InsulinPump’ objects are ‘waiting’ ‘idle’ and ‘idle’ respectively.

The state transition pattern of the ‘:Controller’ object is different in both scenarios and that’s why two state diagrams are supplied for this object and these are shown in Figure 6.13(a) and Figure 6.13(b).

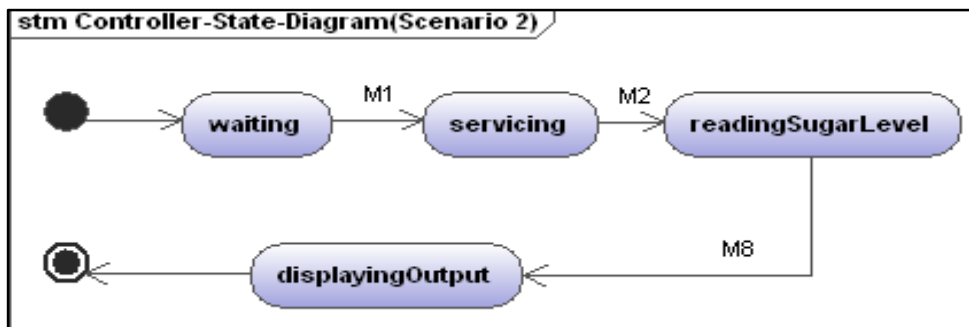
The state diagrams for objects ‘:Sensor’ and ‘:InsulinPump’ are shown in Figure 6.13(c) and Figure 6.13(d) respectively.

The ‘Event-Sequence-State-Transitions’ for scenario 1 is shown in Table 6.22. The ‘Event-Sequence’ for scenario 1 as shown in Table 6.20 and the state state diagrams for the objects ‘:Controller’, ‘:Sensor’ and ‘:InsulinPump’ as shown in Figure 6.13(a), Figure 6.13(c) and Figure 6.13(d) respectively, are used as inputs in the identification of the ‘Event-Sequence-State-Transitions’ for scenario 1.

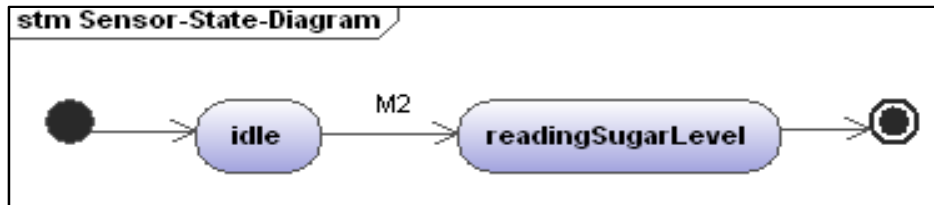
The ‘Event-Sequence-State-Transitions’ for scenario 2 is shown in Table 6.23. The ‘Event-Sequence’ for scenario 2 as shown in Table 6.21 and the state state diagrams for the objects ‘:Controller’, ‘:Sensor’ and ‘:InsulinPump’ as shown in Figure 6.13(b), Figure 6.13(c) and Figure 6.13(d) respectively, are used as inputs in the identification of the ‘Event-Sequence-State-Transitions’ for scenario 2.



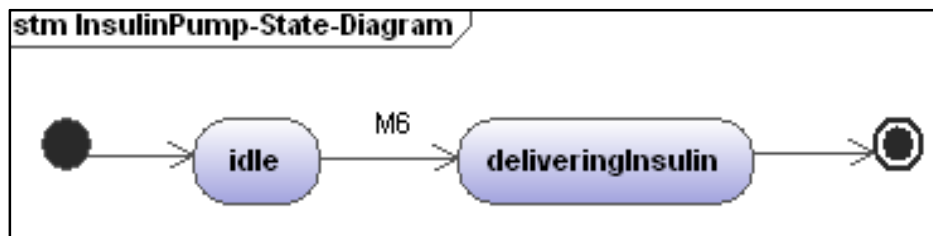
(a) State Diagram of the 'Controller' Object For Scenario 1



(b) State Diagram of The 'Controller' Object For Scenario 2



(c) State Diagram of the 'Sensor' Object



(d) State Diagram of the 'InsulinPump' Object

Figure 6.13: State Diagrams For The Participating Object of IDS System

Table 6.22: ‘Event-Sequence-State-Transitions’ for Scenario 1 of IDS Application

Event	Logical-Time	:Controller	:Sensor	:InsulinPump
e1	1	waiting	idle	idle
e2	2	servicing	idle	idle
e3	3	readingSugarLevel	idle	idle
e4	4	readingSugarLevel	readingSugarLevel	idle
e5	5	readingSugarLevel	idle	idle
e6	6	readingSugarLevel	idle	idle
e7	7	computingInsulin	idle	idle
e8	8	computingInsulin	idle	idle
e9	9	computingInsulin	idle	idle
e10	10	computingInsulin	idle	idle
e11	11	deliveringInsulin	idle	idle
e12	12	deliveringInsulin	idle	deliveringInsulin
e13	13	deliveringInsulin	idle	idle
e14	14	deliveringInsulin	idle	idle

Table 6.23: ‘Event-Sequence-State-Transitions’ for Scenario 2 of IDS Application

Event	Logical-Time	:Controller	:Sensor	:InsulinPump
e1	1	waiting	idle	idle
e2	2	servicing	idle	idle
e3	3	readingSugarLevel	idle	idle
e4	4	readingSugarLevel	readingSugarLevel	idle
e5	5	readingSugarLevel	idle	idle
e6	6	readingSugarLevel	idle	idle
e15	7	displayingOutput	idle	idle
e16	8	displayingOutput	idle	idle
e17	9	displayingOutput	idle	idle
e18	10	displayingOutput	idle	idle

Step IV: Perform ‘Message-Errors-Effects-Analysis’ For Each Scenario

The ‘Message-Errors-Effects-Analysis’ for scenario 1 and scenario 2 are shown in Table 6.24 and Table 6.25 respectively.

Table 6.24: Message-Errors-Effects-Analysis for Scenario 1 of IDS Application

Message#	Message-Errors	Effects													
		e1	e2	e3	e4	e5	e6	e7	e8	e9	e10	e11	e12	e13	e14
M1	!(M1)		controller != servicing	controller != readingSugarLevel	sensor != readingSugarLevel	sensor !=idle		controller != computingInsulin				controller != deliveringInsulin	InsulinPump != deliveringInsulin	InsulinPump !=idle	
M2	!(M2)			controller != readingSugarLevel	sensor != readingSugarLevel	sensor !=idle		controller ^= computingInsulin (M4)				controller ^= deliveringInsulin (M4,M6)	InsulinPump ^= deliveringInsulin (M4,M6)	InsulinPump !=idle (M4,M6)	
M3	(M3)'							controller ^= computingInsulin				controller ^= deliveringInsulin	InsulinPump ^= deliveringInsulin	InsulinPump ^=idle	
M4	!(M4)							controller != computingInsulin				controller ^= deliveringInsulin (M6)	InsulinPump ^= deliveringInsulin (M6)	InsulinPump ^=idle (M6)	
M5	(M5)'											controller ^= deliveringInsulin	InsulinPump ^= deliveringInsulin	InsulinPump ^=idle	
M6	!(M6)											controller != deliveringInsulin	InsulinPump != deliveringInsulin	InsulinPump !=idle	
M7	(M7)TF														
	(M7)FT														

Table 6.25: Message-Errors-Effects-Analysis For Scenario 2 of IDS Application

Message#	Message-Errors	Effects									
		e1	e2	e3	e4	e5	e6	e15	e16	e17	e18
M1	!(M1)		controller != servicing	controller != readingSugarLevel	sensor != readingSugarLevel	sensor !=idle		controller != displayingOutput			
M2	!(M2)			controller != readingSugarLevel	sensor != readingSugarLevel	sensor !=idle		controller ^= displayingOutput (M8)			
M3	(M3)'							controller ^= displayingOutput (M8)			
M8	!(M8)							controller != displayingOutput			
M9	!(M9)										

6.4.3 Analysis of Results

The application of the proposed SFMEA approach enhances the results of the SFTA approach discussed and presented in Chapter 5. The proposed SFMEA approach helps the analyst in the following two ways

- (i) To investigate the effects of the errors associated with not only the state-transition messages (messages, which result in the state change of a component) but with other messages, also where no component is changing its state. Recall that in the SFTA approach of Chapter 5, only the errors associated with the state-transition messages are considered.
- (ii) To construct the software fault tree for the hazardous-state where a state of a component is changed erroneously.

Consider the ‘Message-Errors-Effects-Analysis’ for RTCS application scenarios as shown in Table 6.17 and Table 6.18. The message ‘M1’ is not changing the state of any of the components as shown in Figure 6.10. But the effects of the error associated with it i.e. ‘!(M1)’ (message ‘M1’ is not sent) is shown in Table 6.17 and Table 6.18. In the same way, the message ‘M4’ is also is not changing the state of any of the components as shown in Figure 6.10. The errors associated with this message are ‘(M4)FT’ and ‘(M4)TF’ and their effects are shown in Table 6.17 and Table 6.18. Similarly, the ‘Message-Errors-Effects-Analysis’ for IDS application scenarios, as shown in Table 6.24 and Table 6.25, record the effects of the errors associated with the messages ‘M3’, ‘M5’ and ‘M7’ and no component is changing its state during the sending of these messages (see Figure 6.13).

Consider the ‘Message-Errors-Effects-Analysis’ for scenario 2 of the RTCS application as shown in Table 6.18. The error ‘!(M3)’ erroneously changes the state of the signal component to green provided the message M6 is sent in case of this error (i.e. ‘signal[^]=green(M6)’). The effect entry ‘signal[^]=green(M6)’ also appears for the error ‘(M4)FT’. The fault tree for the hazardous-state ‘signal[^]=green’ is drawn by ORing the message-related errors and the messages represented in these rows. In a similar manner, the fault tree for the hazardous-state ‘InsulinPump[^]=deliveringInsulin’ for an IDS application can be drawn from the ‘Message-Errors-Effects-Analysis’ as shown in Table 6.24.

6.5 COMPARATIVE ANALYSIS

The presented SFMEA approach is an attempt towards developing a fully automated tool. There are other approaches also that have been explored for UML such as HazOp (Hazard and Operability Study) (Lu et al, 2005) but its application is manual. The main strength of the presented approach is that it is automated and can be applied even if the dysfunctional behavior of the participating objects is not known (David et al, 2008). The message-related errors are identified automatically in the presented approach.

The developed approach has scope for several improvements. Some of the situations that need to be further improved and are considered as scope for further work are enumerated below.

- (i) Support for ‘par’ and ‘loop’ interaction operators: The presence of ‘par’ operator may complicate the building of MSCFG in Step II. Similarly, presence of loops complicates the scenario extraction process because in a loop a message sequence may repeat any number of times.
- (ii) Number of Message-Related Errors Addressed: Currently the algorithm can handle only a limited number of message-related errors. Especially, the timing related message errors where a message arrives either too late or too early, will require time as another parameter in the SFMEA approach.
- (iii) Only one message-related error is considered as active at any given point of time. The approach in the present form can be applied only to sequential systems and not for concurrent systems where multiple errors may occur at the same time.

Software Reliability Prediction for Use-Cases

This chapter presents the use of the SFMEA and SFTA approaches, developed for object-oriented use-cases, to predict the software reliability of a given use-case functionality during the requirements analysis phase. The reliability of each scenario of the given use-case is predicted from the constructed generalized fault tree for scenario failures using the probability of occurrence value of each basic erroneous event.

Software reliability is defined as ‘the probability of failure-free operation for a specified period of time in a specified environment’ (IEEE-STD-729-1991, 1991). Software reliability is one the four key dependability attributes, namely Availability, Reliability, Safety and Security for safety-critical systems. The software failure is defined as, ‘deviation of the delivered service from compliance with the specification’. Correct prediction of the probability of occurrence of various software-related errors is the key to the estimation of the reliability of a software system and to estimate and predict the failure rates of software systems.

Currently, the software reliability is generally estimated after the implementation phase by subjecting the software code to reliability evaluation. This is too late for safety critical software systems. The current research efforts are therefore focused towards the prediction of the reliability of software systems during the early phases such as in requirements analysis and design phases. An overview of the early software estimation approaches is given in the next section.

7.1 EARLY SOFTWARE RELIABILITY ESTIMATION APPROACHES

Meng (Meng et al, 2000) proposed petri-net based method for early-stage software reliability estimation. The limitation of their work is that it requires the hierarchical view of the software system and cannot work with large systems.

Singh (Singh et al, 2001) and Cortellessa (Cortellessa et al, 2002) proposed a Bayesian approach based reliability prediction method for component based software systems. The method requires the annotation of Unified Modeling Language (UML) models for reliability prediction. The problem with the approach is that it requires models from two

phases namely requirements analysis (use-cases), design (sequence and deployment diagrams) phases.

The Scenario Based Reliability Analysis (SBRA) method proposed by Yacoub (Yacoub et al, 2004) estimates the reliability of a software system from the reliabilities of its components. It is assumed that the reliability of each component is known in advance.

Tripathi and Mall (Tripathi and Mall, 2005) proposed an early reliability estimation technique (ERAT) for use-cases based on reliability block diagrams (RBD). The RBD is used to model the relationship between use-case and its scenarios.

Kong (Kong et al, 2007) proposed a binary decision diagram (BDD) based early reliability prediction method using a cause-effect graphing analysis (CEGA) technique. The CEGA technique is used to identify the defects in a software requirements specification document and the impacts of these defects in the system are assessed using BDD.

Kundu and Samanta (Kundu and Samanta, 2007) proposed a three step approach to assess reliability of a system using the use-case model. In the first step, a given use-case model is converted into a system sequence diagram. The second step converts a sequence diagram into a use case graph (UCG). The reliability metric of each use-case scenario is determined in the third step.

Mohanta's work (Mohanta et al, 2010) takes into account the information about the operational profile and usage frequency of each use-case functionality to estimate the reliability of the individual use-case as well as its associated scenarios.

The above four reliability prediction approaches for use-cases require the results of design phase for reliability prediction. None of the above-mentioned approaches puts emphasis on forecasting the errors that can occur during the realization of the use-case. The proposed SFTA and SFMEA analysis of use-cases provide a simple solution for early reliability estimation. The results of SFTA/SFMEA approaches build a fault tree or database of event-related errors that can be experienced by the system during the use-case execution. Coupled with occurrence rate to each identified event-related error, this error database can be used to predict the reliabilities of various use-cases and their associated scenarios. This is explained in the following sections of this chapter and is illustrated with examples.

7.2 PROPOSED SOFTWARE RELIABILITY ESTIMATION APPROACH FOR USE-CASES

7.2.1 Assumptions

- (i) The operational profile(OP) (usage frequency of each use-case) of each use-case functionality and its each scenario is available
- (ii) The error occurrence rates of various event-related errors are also known.

7.2.2 Operational Profile of a Use-Case

The operational profile (OP) of a software system is a quantitative characterization of how the software will be used and is, therefore, essential in any Software Reliability Engineering (SRE) application (Musa, 1993). The concept of operational profile is explained below by taking an arbitrary use-case model as shown in Figure 7.1 with three use-cases, namely 'doA', 'doB' and 'doC' and one actor, namely 'X'.

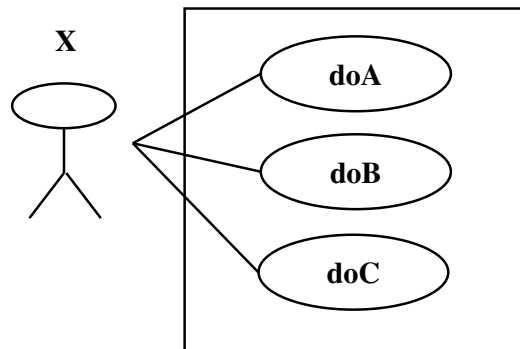


Figure 7.1: An Example Use-Case Model

The usage frequencies of the various use-cases constitute the operation profile of the system. Some use-cases are executed more frequently than the others. The failure occurrence rate is more in more frequently used operations (use-cases) than the operations that are less frequently used. Assume P_1 , P_2 and P_3 are the execution probabilities of use-cases 'doA', 'doB' and 'doC' respectively. The summation of these probabilities i.e. $\sum P_i$ (where $i = 1,2,3$) is 1.

Suppose the actor 'X' uses or accesses the system 100 times and 'doA' operation is accessed 70 times, 'doB' is accessed 20 times and 'doC' operations is accessed 10 times. Then the usage frequency for 'doA', 'doB' and 'doC' use-cases are 70%, 20% and 10%, respectively. These usage frequency values are used to compute the execution

probability values for the use-cases. The execution probability of ‘doA’, ‘doB’ and ‘doC’ use-cases are ‘0.70’, 0.20’ and ‘0.10’, respectively.

7.2.3 The Proposed Reliability Estimation Approach for Use Cases

As discussed in previous chapters that a use-case represents a functionality offered by the software system and it can have any number of unique execution paths known as scenarios. In order to estimate the reliability of a safety critical software system, it is important and mandatory to predict the reliabilities of the use-cases and their scenarios. The reliability of a given use-case functionality and its various scenarios depend upon the operational profile of the system.

The textual description of a given use-case functionality and the state diagrams of the participating components are used as inputs in the proposed approach and is divided into five steps to predict the reliability of a given use-case functionality. These steps are explained below:

Step I Applying SFMEA Approach

The application of SFMEA approach as presented in Chapter 4 gives the information about (i) the number of scenarios in a given use-case, (ii) the event-related errors that can occur in each scenario and (iii) the effects of each event-related error in the system.

Step II Predicting the Usage Frequency of Each Scenario of the Use-case

The second step is to predict the value of the usage-frequency (i.e. The execution probability) for each scenario of the use-case. Out of the number of scenarios of a use-case, one of the scenarios is considered as *main-scenario* and other scenarios are characterized as *exceptional* or *alternative* scenarios. The *main-scenario* represents the execution path of the use-case that is executed more frequently than the other execution paths. Hence, the usage frequency of the *main-scenario* is generally assigned higher than the other scenarios. The analyst’s domain expertise, experience and knowledge of the system play an important role in determining the values of these usage frequencies.

Step III Predicting the ‘Probability of Occurrence’ Value to each Event-Related Error

In the third step, the ‘probability of occurrence’ value for each event-related error of the use-case functionality is predicted. The application of SFMEA approach, in Step I,

identifies all the event-related errors and their effects. It is assumed that the occurrence of event-related errors is independent of each other. A failure in a scenario is said to have occurred if any one of its associated event-related error occurs. If an event-related error 'E' has a probability of occurrence value as '0.002' then it means the error 'E' can occur 2 times in 1000 executions of the event. The experience and domain expertise of the analyst plays an important role in the prediction of this value.

Recall that in the application of SFMEA approach in the use-case based requirement analysis phase (Chapter 4), two types of event-related errors, namely (i) stop type and (ii) propagating type, are considered. A stop type event-related error is a software control error and can occur during the execution any event. It is assumed in the approach that the error occurrence rate ('probability of occurrence') of the stop-type errors is uniform for all events. For propagating type of event-related errors, the 'probability of occurrence' values are assigned by taking the past experience into consideration.

Step IV Computing the Reliability for Each Scenario

The fourth step computes the reliability of each scenario by constructing a generalized fault tree for the scenario failure by taking 'Event-Errors-Effects-Analysis' (results of the SFMEA approach in Step I) of the scenario as an input. The event-related errors are mapped against their erroneous state level effects. The fault tree construction process is as follows:

- (i) If an erroneous state level effect 'X' is caused by more than one event-related error then all the event-related errors responsible for 'X' are joined by an 'OR' gate.
- (ii) If an event-related error 'E' causes more than one erroneous state level effect then 'E' is used as basic erroneous event only for the erroneous state level effect that occurs earlier. The output of the first effect is used as an input for the next level erroneous state level effect and so on.

The fault tree is constructed using a FaultCAT tool (FaultCAT, 2003). Using the fault tree for scenario failure, the probability of scenario failure is computed by using the 'probability of occurrence' values of the basic erroneous events. If the basic erroneous events are joined via an 'OR' gate then their 'probability of occurrence' values are added to get the failure probability value of the next intermediate event of the fault tree. If the

basic erroneous events are joined via an ‘AND’ gate then their ‘probability of occurrence’ values are multiplied to get the failure probability value of the next intermediate event of the fault tree. This process is recursively applied till the probability of a scenario failure (top event) is obtained.

The reliability of the scenario is computed using the formula: ‘ $1 - \text{fail}(S)$ ’ where ‘ $\text{fail}(S)$ ’ is the probability of the scenario failure.

Step V Computing the Reliability of the Use-Case

The reliability of a given use-case functionality ‘ U ’ is computed by using the formula proposed by Mohanta (Mohanta et al, 2010):

$$R(U) = 1 - \prod_{i=1}^M (1 - \text{rel}(S_i)) \times p_i$$

Where $R(U)$ is the reliability of the use-case ‘ U ’

$\text{rel}(S_i)$ is the reliability of the i^{th} scenario of the use-case

M is the number of the scenarios in the use-case

p_i is the usage frequency of the i^{th} scenario

The application of the approach is illustrated for two case study applications, namely ‘Insulin Delivery System’ (IDS) and ‘Rail Track Door Control System’ (RTCS) discussed in the previous chapters is given in the next sections.

7.3 MOTIVATING EXAMPLE 1: INSULIN DELIVERY SYSTEM

The detailed description of the Insulin Delivery System (IDS) case study is given in Chapter 4. The application of five steps of the reliability computation approach is given below:

Step I: Applying SFMEA approach

The application of the SFMEA analysis of IDS (Chapter 4) for use-case ‘Deliver Insulin’ results in two scenarios and the event sequences of these two scenarios are shown in Table 4.6 and Table 4.7, respectively. The ‘Event-Errors-Effects-Analysis’ of the two scenarios are shown in Table 4.11 and Table 4.12.

Step II: Predicting the usage frequency to each scenario of the use-case

The purpose of the system is to deliver an insulin to the patient as and when required. There is only one use-case, namely 'Deliver-Insulin' in this case study. Only this use-case gets executed as long as the system is in operation. Therefore, the execution probability of this use-case is 1. However, there are two scenarios in this use-case. The scenario 1 (Table 4.6) represents the situation when an insulin is delivered and the second scenario (Table 4.7) represents the situation when an insulin is not delivered because the sugar level in the patient's body is within acceptable level. The scenario 2 is the main-scenario and scenario 1 is an exceptional or an alternative scenario. It is assumed that the 'Deliver Insulin' insulin use-case is executed 24 times a day (from 9:00 AM to 9:00 PM after every 30 minutes duration) and scenario 2 is executed 18 times (with usage frequency of 75%) and scenario 1 is executed 6 times (with usage frequency of 25%). So, the execution probability for scenario 2 is arbitrarily assigned as '0.75' where as the execution probability for scenario 1 is assigned as '0.25'. [Note that the sum of execution probabilities of the scenarios of a use-case is equal to the execution probability of the use-case].

Step III: Predicting the 'probability of occurrence' value for each event-related error

The event-errors for IDS system are shown in Table 4.10. The probability of occurrence values assigned for each event-related error are assumed as shown in Table 7.1. The system is safety-critical and it is assumed that the software used to control the system should be highly reliable. Therefore, the software related errors are assigned lower 'probability of occurrence' value than the hardware related errors. Each stop type error (ER1, ER2, ER5, ER8, ER10 and ER12) is basically a software related error and assigned a 'probability of occurrence', say '0.005' (i.e. occurrence rate of only 0.5%). The errors 'ER3', 'ER4', 'ER11' belongs to hardware related errors and are assigned a 'probability of occurrence', say '0.010' (i.e. 1% occurrence rate) which is higher than the software related errors. The system is considered highly reliable from computational dimension and there is very less chance of any computational type error. That's why the 'probability of occurrence' value assigned to errors 'ER6', 'ER7' and 'ER9' is '0.001' (i.e. occurrence rate of 0.1%).

Table 7.1: Assumed Probability of Occurrence of Event-Related Errors of IDS Application

Event#	Error#	Type	Error Description	Probability of Occurrence
E0	ER1	1	Event E0 fails to execute	0.005
E1	ER2	1	Event E1 fails to execute	0.005
	ER3	1	sensor failure	0.010
	ER4	2	The sensor reads the wrong sugar value	0.010
E2	ER5	1	Event E2 fails to execute	0.005
	ER6	2	Event E2 is true, but evaluated as false	0.001
	ER7	2	Event E2 is false, but evaluated as true	0.001
E3	ER8	1	Event E3 fails to execute	0.005
	ER9	2	The system computes wrong insulin dose	0.001
E4	ER10	1	Event E4 fails to execute	0.005
	ER11	1	Insulin pump fails to deliver Insulin	0.010
E5	ER12	1	Event E5 fails to execute	0.005

Step IV: Constructing fault tree and predicting reliability of each scenario

Using the ‘Event-Errors-Effects-Analysis’ as shown in Table 4.11 and Table 4.12, the fault trees for two scenarios are constructed. The errors that can occur in scenario 1 are ER1, ER2, ..., ER11. The errors that can occur in scenario 2 are ER1, ER2, ER3, ER4, ER5, ER6, ER7 and ER12.

(i) Reliability prediction of scenario 1

The fault tree constructed for scenario is shown in Figure 7.2. The probability of failure of scenario 1 is computed from the fault tree by computing the ‘probability of occurrence’ values for the various events as follows:

$P(X)$ = probability of occurrence of 'X' where 'X' is either a basic erroneous event or an intermediate state level effect or root node of the fault tree. Using this, various probabilities are obtained as follows:

$$P(\text{Insulin-Controller} \neq \text{servicing}) = P(\text{ER1}) = 0.005$$

$$\begin{aligned} &P(\text{Insulin-Controller} \neq \text{ReadingSugarLevel} \text{ AND } \text{Sugar-Sensor} \neq \text{ReadingSugarLevel}) \\ &= P(\text{Insulin-Controller} \neq \text{servicing}) + P(\text{ER2}) + P(\text{ER3}) \\ &= 0.005 + 0.005 + 0.010 \\ &= 0.020. \end{aligned}$$

$$\begin{aligned} &P(\text{Insulin-Controller} \neq \text{ComputingInsulinDose}) \\ &= P(\text{Insulin-Controller} \neq \text{ReadingSugarLevel} \text{ AND } \text{Sugar-Sensor} \neq \text{ReadingSugarLevel}) \\ &\quad + P(\text{ER5}) + P(\text{ER8}) \\ &= 0.020 + 0.005 + 0.05 = 0.030. \end{aligned}$$

$$\begin{aligned} &P(\text{Insulin-Controller} \neq \text{DeliveringInsulin} \text{ AND } \text{Insulin-Pump} \neq \text{DeliveringInsulin}) \\ &= P(\text{Insulin-Controller} \neq \text{ComputingInsulinDose}) + P(\text{ER10}) + P(\text{ER11}) \\ &= 0.030 + 0.005 + 0.010 \\ &= 0.045 \end{aligned}$$

$$\begin{aligned} &P(\text{Insulin-Controller} \neq \text{ComputingInsulinDose}) \\ &= P(\text{ER4}) + P(\text{ER7}) + P(\text{ER9}) \\ &= 0.010 + 0.001 + 0.001 \\ &= 0.012 \end{aligned}$$

$$\begin{aligned} &P(\text{Insulin-Controller} \neq \text{DeliveringInsulin} \text{ AND } \text{Insulin-Pump} \neq \text{DeliveringInsulin}) \\ &= P(\text{Insulin-Controller} \neq \text{ComputingInsulinDose}) \\ &= 0.012. \end{aligned}$$

$$\begin{aligned} &P(\text{Failure in Scenario 1}) \\ &= P(\text{Insulin-Controller} \neq \text{DeliveringInsulin} \text{ AND } \text{Insulin-Pump} \neq \text{DeliveringInsulin}) \\ &\quad + P(\text{Insulin-Controller} \neq \text{DeliveringInsulin} \text{ AND } \text{Insulin-Pump} \neq \text{DeliveringInsulin}) \\ &= 0.045 + 0.012 \\ &= 0.057 \end{aligned}$$

Thus, the computed reliability of the scenario 1 = '1 - 0.057' = '0.943'.

(ii) Reliability prediction of scenario 2

The fault tree constructed for scenario 2 is shown in Figure 7.3. Following the steps as above, the computed reliability of scenario 2 is '0.959'.

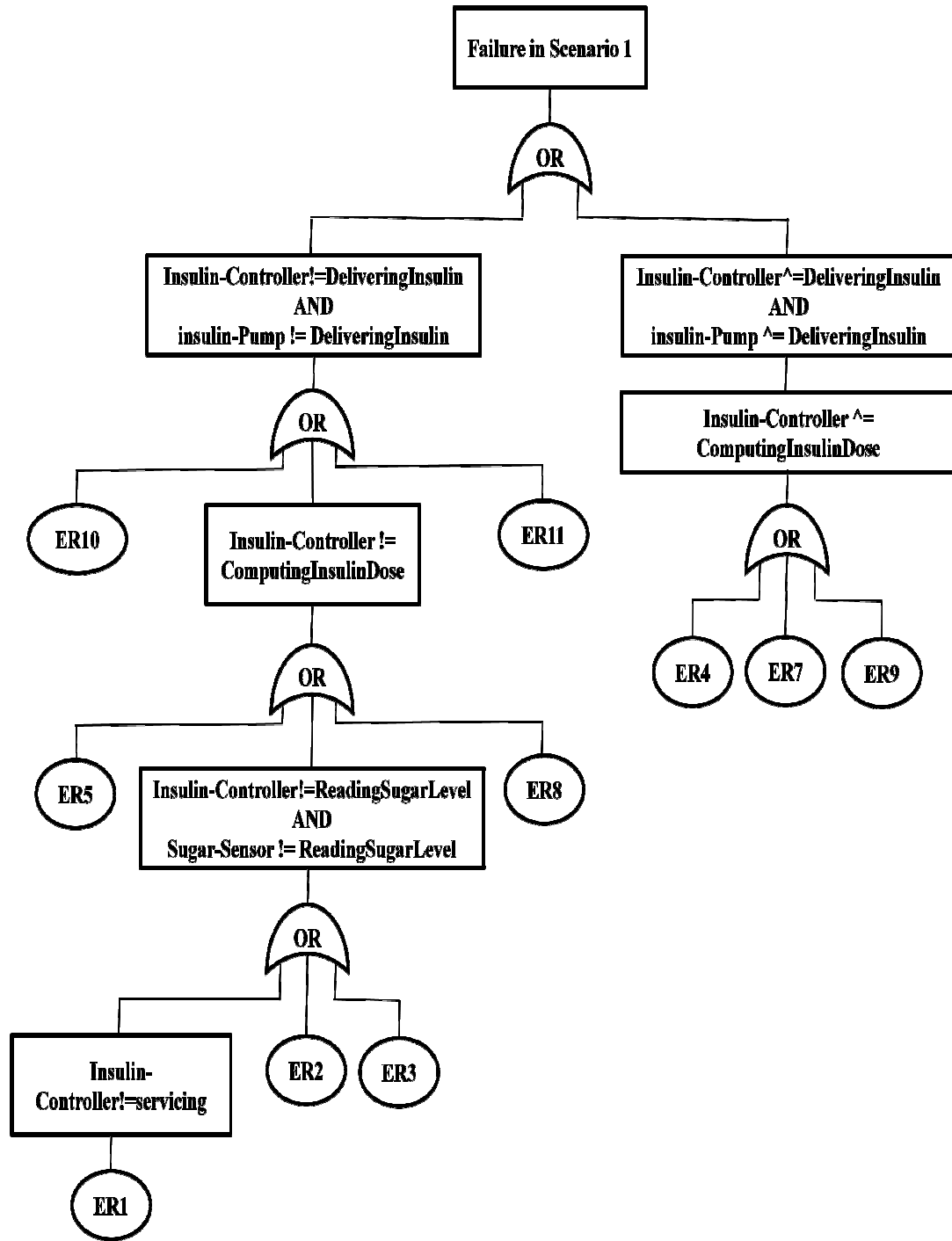


Figure 7.2: Fault Tree for Failure of Scenario 1 of IDS Application

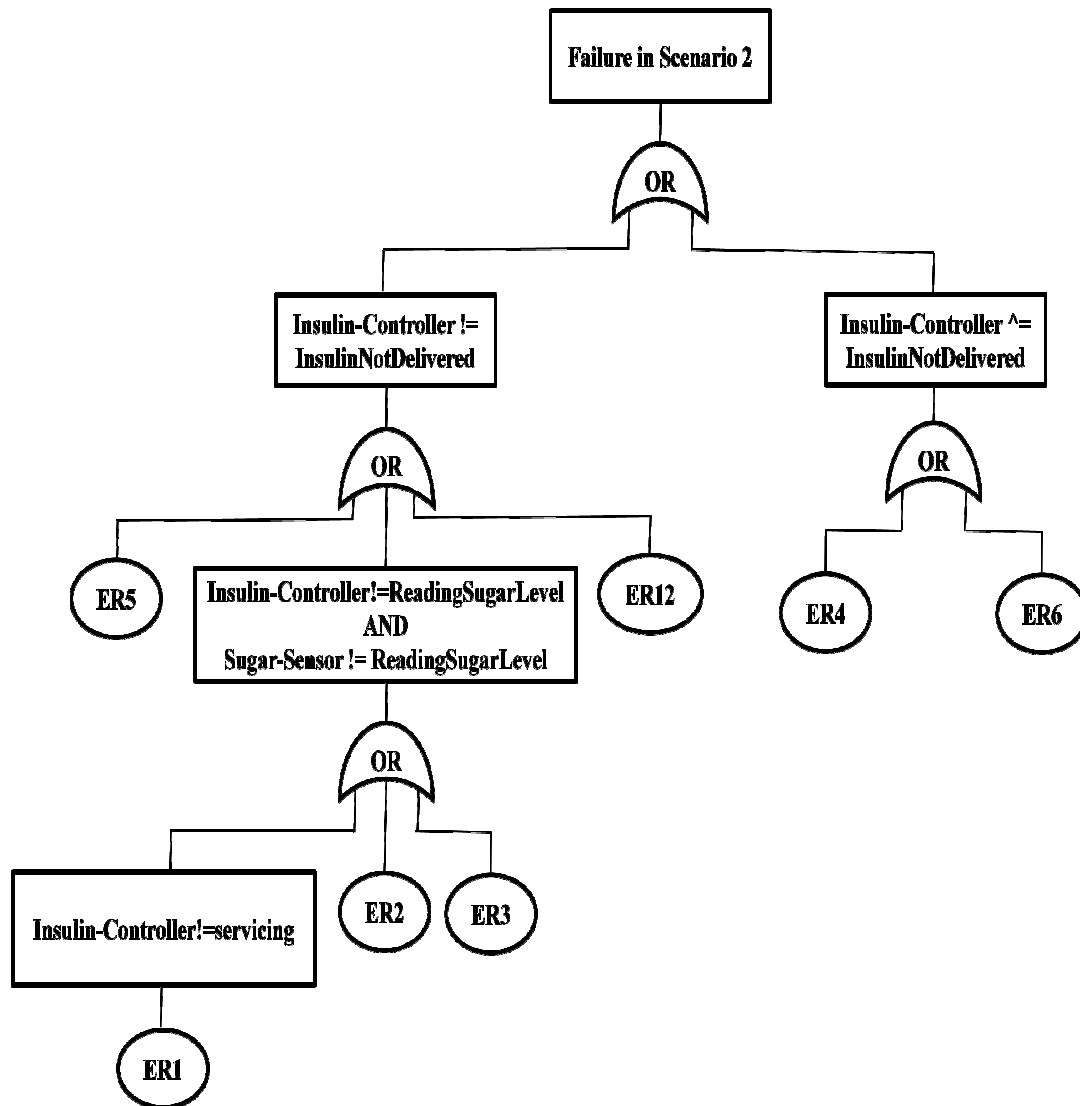


Figure 7.3: Fault Tree for Failure of Scenario 2 of IDS Application

Step V: Computing the reliability of the 'Deliver-Insulin' use-case

The reliability of 'Deliver Insulin' use-case is calculated using the following values

Reliability of Scenario 1 = 0.943

Execution Probability of Scenario 1 = 0.25

Reliability of Scenario 2 = 0.959

Execution Probability of Scenario 2 = 0.75

$$\begin{aligned}
 R(\text{'Deliver Insulin'}) &= '1 - \{(1 - 0.943) \times 0.25\} \times \{(1 - 0.959) \times 0.75\}' \\
 &= 0.9995
 \end{aligned}$$

Therefore, the reliability of the 'Deliver Insulin' use-case is '0.9995'.

7.4 MOTIVATING EXAMPLE 2: RAIL TRACK DOOR CONTROL SYSTEM

The detailed description of the Rail Track Door Control System (RTCS) case study is given in Chapter 3. The applications of five steps of the approach is given below.

Step I: Applying SFMEA approach on the inputs

There are two scenarios in this application and the ‘Event-Sequences’ for scenario 1 and for scenario 2 are shown in Table 3.22 and Table 3.23 respectively.

The ‘Event-Errors-Effects-Analysis’ for scenario 1 and scenario 2 are shown in Table 4.14 and table 4.15 respectively.

Step II: Predicting the usage frequency of each scenario of the use-case

The purpose of the system is to close/open the track door whenever the train is to arrive/depart. There is only one use-case in this functionality namely ‘close track door’. The probability of execution of ‘close-track-door’ is 1 because it is executed whenever an interrupt is generated by the track sensors. However, there are two scenarios in this use-case functionality. The main-scenario of this case study application represents the situation when the track door is closed successfully. The exceptional or alternative scenario is executed when the system detects an error in the track door and issues track door failure. The execution probability for scenario 1 is assumed as ‘0.95’ where as the execution probability for scenario 2 is assumed as ‘0.05’.

Step III: Predicting the probability of occurrence value for each event-related error

The ‘Event-Errors’ identified for this application are shown in Table 4.13. The probability of occurrence values assigned to each event-related error is shown in Table 7.2. The rule for selecting the ‘probability of occurrence’ values is same as followed in the previous case study application.

Step IV: Constructing fault tree and predicting the reliability of each scenario

Consider the ‘Event-Errors-Effects-Analysis’ for scenario 1 and scenario 2 as shown in Table 4.14 and table 4.15.

The fault tree constructed for scenario 1 by taking the ‘Event-Errors-Effects-Analysis’ as shown in Table 4.14 as an input is shown in Figure 7.4. The fault tree constructed for

scenario 2 by taking the ‘Event-Errors-Effects-Analysis’ as shown in Table 4.15 as an input, is shown in Figure 7.5. The probability values to each event-related error is assigned as per Table 7.2.

The reliability of scenario 1 of the RTCS application = ‘1 – 0.044’
= ‘0.956’.

The reliability of scenario 2 of the RTCS applications = ‘1 – 0.020’
= ‘0.98’.

Table 7.2: Probability of Occurrence of Event-Related Errors of RTCS Application

Event#	Error#	Type	Error Description	Probability of Occurrence
E1	ER1	1	Event E1 fails to execute	0.005
E2	ER2	1	Event E2 fails to execute	0.005
	ER3	1	Error in Track Door Component	0.010
E3	ER4	1	Event E3 fails to execute	0.005
	ER5	2	Event E3 is False but evaluated as True	0.001
	ER6	2	Event E3 is true but evaluated as False	0.001
E4	ER7	1	Event E4 fails to execute	0.005
	ER8	1	Error in Track_signal Component	0.010
E5	ER9	1	Event E5 fails to execute	0.005
E6	ER10	1	Event E6 fails to execute	0.005

Step V: Computing the reliability of the ‘close-track-door’ use-case

The reliability of scenario 1 = 0.956

The execution probability of scenario 1 = 0.95

The reliability of scenario 2 = 0.98

The execution probability of scenario 2 = 0.05

Thus, the reliability of close-track-door use-case is ‘0.9999’.

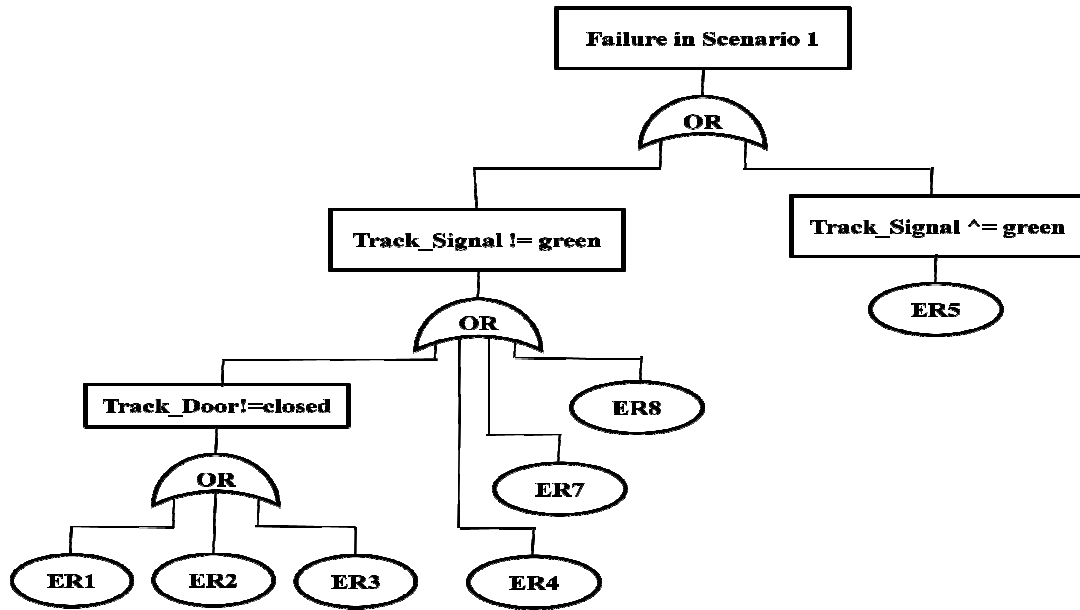


Figure 7.4: Fault Tree for Failure in Scenario 1 of RTCS Application

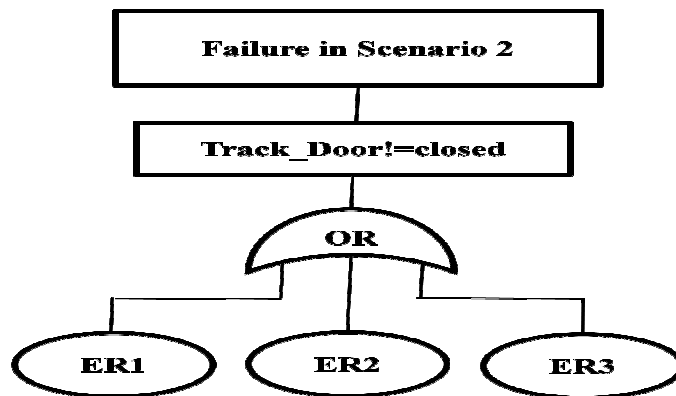


Figure 7.5: Fault Tree for Failure in Scenario 2 of RTCS Application

7.5 ANALYSIS OF RESULTS

The reliability prediction methods, as reported in the works of Singh (Singh et al, 2001) and Cortellessa (Cortellessa et al, 2002), require the conversion of a given use-case functionality into a set of sequence diagrams and it is assumed that the failure probability of each component (i.e. class) involved in the use-case is known in advance. The approach cannot be used if only use-cases are used as inputs because the sequence diagrams are generally drawn during object-oriented design phase. Similarly, the reliability assessment methods proposed by Kundu and Samanta (Kundu and Samanta,

2007) and Yacoub (Yacoub et al, 2004) also require the conversion of a given use-case into a sequence diagram.

The proposed approach is developed for use-case based requirements analysis phase and there is no need to convert a use-case into its corresponding sequence diagram or reliability block diagram.

The method as reported by Tripathi and Mall (Tripathi and Mall, 2005) considers the failure only in the main scenario to predict the reliability of a use-case. However, a use-case functionality can have any number of scenarios and an error can occur in each scenario. The proposed approach considers failures in every scenario to predict the reliability of a given use-case.

The proposed approach computes the reliability of a scenario using the simple FTA approach whereas Mohanta (Mohanta et al, 2010) computes the reliability of a scenario 'R(S); by using the the number of classes required to implement and execute the scenario and the reliability of the each class. The reliability of each class is to be known in order to compute the reliability of the scenario. In object-oriented design process, the classes and their attributes and methods are identified only in design phase. Therefore, Mohanta's method cannot be used until the design phase when all the classes required to execute the scenario and their reliabilities are known.

On the other hand, the proposed approach computes the reliability of a scenario by constructing a generalized fault tree for the scenario in Step IV. The approach can be used even if the required classes to implement the scenario are not known.

7.6 SOFTWARE RELIABILITY PREDICTION AT OBJECT-ORIENTED DESIGN PHASE

The SFMEA approach for object-oriented design phase presented and discussed in Chapter 6 can be used to predict the software reliability of a particular use-case in design phase by improving it so that it can investigate the erroneous effects of every message-related error. The SFMEA approach for use-cases presented in Chapter 4 investigates the effect of every event-related error. On the other hand, only five types of message-related errors are considered in the SFMEA approach for object-oriented design phase. Therefore, in order to estimate the reliability of a given use-case in design phase, the developed SFMEA approach for design phase should consider the effects of every message-related error.

Conclusion and Future Research Directions

This chapter concludes the thesis with a summary of the proposed SFTA and SFMEA approaches for the object-oriented requirements analysis phase and design phase and then outlines the directions for carrying out further research in the area. As introduced in Chapter 1 and analyzed in Chapter 2, the existing SFTA and SFMEA approaches, in the object-oriented based requirements analysis and design phases, are manual, time-consuming and error-prone. The thesis has made key contributions by developing automated and semi-automated SFTA and SFMEA approaches for application in the early phases of object-oriented based requirements analysis and design. The developed algorithms use formal textual use-case description and the state diagrams in the SFTA and SFMEA approaches for object-oriented requirements analysis phase and sequence and state diagrams for the SFTA and SFMEA approaches for object-oriented design phase.

A summary of the developed SFTA and SFMEA approaches is given in the next section

8.1 PROPOSED SFTA AND SFMEA APPROACHES - SUMMARY

A review of the current literature on early software reliability estimation for safety critical software systems clearly demonstrates that the available methods are only for very late stages in the software development cycle and are manual (heavily dependent on expertise), cumbersome, time-consuming and error-prone. This thesis is a minimal attempt to provide a methodology for software reliability estimation at an early stage of software development cycle and automating the process by using only the information on use-cases available at the stage of requirements analysis and design.

The introduction about the basics of the SFTA and SFMEA approaches and various UML models such as use-cases, sequence and state diagrams is presented in Chapter 1. The research objectives of the thesis are established based upon the problems faced during the applications of the SFTA and SFMEA approaches in object-oriented based requirements analysis and design phases.

The critical review of literature on use of the SFTA and SFMEA applications is done in two software life-cycle phases, namely requirements analysis and design phases and the summary of the findings is presented in Chapter 2. Based upon the findings, the current research gaps especially in the applications of the SFTA and SFMEA approaches in object-oriented based requirements analysis and design phases are identified. These research gaps are addressed in subsequent chapters.

In the early prediction of software reliability for safety critical systems, the existing approaches have limitations. The first approach proposed in this thesis for modeling the system is using a software fault tree approach. In chapter 3, an approach is presented to address the first research gap (automating/semi-automating the application of the SFTA approach in use-case based requirements analysis phase). The approach is applicable as early as in use-case based requirements analysis. The developed SFTA approach is based on integrating the features of use-cases and state diagrams to automatically generate a software fault tree for a hazardous state of the system. The approach first builds the correct state of the system by mapping the events of the given use-case description against the states of the participating components and then constructs the software fault tree for the selected hazardous state of the system. The approach is automated, efficient and scalable. The limitation of the approach is that it requires the hazardous-state representation in terms of states of the participating components.

The second approach proposed in this thesis for the failure analysis of safety critical systems is using a software failure modes and effects analysis approach. The SFMEA approach developed for use-case based requirements analysis process is discussed in Chapter 4. The SFTA approach of Chapter 3 only considers state transition errors i.e. the errors, which prevent a component from making their expected state transitions. The errors occurring during the execution of other events (where no component is changing its state) are not considered in the SFTA approach of Chapter 3. In order to overcome this limitation, a semi-automated SFMEA approach is developed and introduced in use-case based requirements analysis process. The developed SFMEA approach is discussed in Chapter 4. The approach is semi-automatic because of the propagating type errors for normal events are to be identified manually.

The SFTA approach developed for object-oriented design phase is semi-automatic and is presented and discussed in Chapter 5. The approach takes a sequence diagram and the state diagrams of the participating objects as inputs. The approach maps the events of

various messages of the sequence diagram against the state of the participating objects. The approach is semi-automatic because (i) the approach requires the proper tagging of the sequence diagram and sequence diagram represents the functionality of only one use-case scenario and (ii) a specific naming pattern for naming the states of the participating objects is to be used.

The SFMEA approach developed for object-oriented design phase and presented in Chapter 6, is automated. This approach is developed to overcome the limitations of the SFTA approach of Chapter 5. In the developed SFMEA approach, the sequence diagram can represent multiple scenarios of a given use-case functionality and there is no restriction on the naming the states of the participating objects. The limitation of the approach is that only select categories of message errors are supported in the approach.

The existing software reliability prediction approaches for use-cases has a limitation that these approaches require the results of the successive phases in order to estimate the reliability of a given use-case functionality. In order to overcome this limitation, a novel SFMEA and SFTA based approach is developed to predict the software reliability of a given use-case functionality. The approach does not require any results from the successive phases and can be used during the use-case based requirements analysis phase itself. In the proposed approach, the SFMEA technique is first applied on the given use-case functionality and then software reliability of a use-case is predicted by constructing a generalized software fault tree for the use-case failure. The advantage of the approach is that it can be applied either in object-oriented based requirements analysis or in design phase.

8.2 FUTURE RESEARCH DIRECTIONS AND RECOMMENDATIONS

The developed SFTA and SFMEA approaches have some limitations and these are discussed at the end of Chapter 3,4,5 and 6. The developed SFTA and SFMEA approaches, as discussed and presented in Chapter 3 and Chapter 4 respectively, suffer from the following limitations.

- (i) The approaches in the present form does not handle the case where the participating components are experiencing concurrent state transitions.
- (ii) The timing related errors are not addressed neither in the SFTA nor in the SFMEA approach.

The SFTA and SFMEA approaches developed for object-oriented design phase, presented, and discussed in Chapter 5 and Chapter 6 respectively also suffer from the above-mentioned limitations. Moreover, all the developed approaches depend heavily on the correctness and completeness of the required inputs.

In order to overcome the above-mentioned limitations, the developed approaches are to be made more versatile to provide for the following features

(a) Checking the correctness and completeness of the inputs before applying the algorithms

The developed SFTA and SFMEA approaches for object-oriented based requirements analysis and design phases are developed under the assumption that the inputs supplied in the approaches are correct and complete. The approaches do not check the consistency of the inputs before the application of the approach. The support for this feature requires the development of correctness and completeness criteria for a given use-case functionality.

(b) Handling of Concurrent State Transitions

The proposed SFTA and SFMEA approaches for object-oriented requirements analysis and design phases do not provide the support to handle concurrency. The approaches are used or applied on the sequential applications where it is possible to know the order (i) of various messages (in sequence diagrams), (ii) of state transitions (in case of state diagrams) and (iii) of the occurrence of various events. In concurrent systems, the individual messages or message sequences may be sent in parallel, the state transitions may occur in parallel (known as orthogonal states) and in such situations, it is difficult to know the above-mentioned three facts.

(c) Development of SFTA and SFMA Assisted UML tool

The developed approaches are tested by using a UML tool namely Altova UModeler (UModel, 2013) tool and a fault tree creation and analysis tool namely FaultCAT (FaultCAT, 2003). The sequence and state diagrams are drawn using Altova tool and fault trees are constructed using FaultCAT tool. There does not exist a single UML tool that supports the features similar to the FaultCAT tool.

(d) Handling of Timing Related Errors

The timing related errors are not considered in any of the SFTA and SFMEA approaches discussed in Chapter 3 to Chapter 6. The concept of ‘logical time’, employed in all the approaches, merely tells which event occurs earlier than the others. It does not give any information about how much physical time an event actually takes to execute or respond.

(e) Using SFMEA and SFTA approaches for Software Reliability Prediction in Object-Oriented Design Phase

An automated SFMEA approach developed for object-oriented design phase and presented in Chapter 6 only considers a limited number of message-related errors. However, to predict the software reliability in object-oriented design phase, every message-related error is to be accounted for and their effects have to be analyzed. So, in future efforts should be made to include the effects of every message-related error in the developed SFMEA approach (of Chapter 6) so that the new augmented approach can be used to predict the software reliability in object-oriented design phase in a similar way as used in object-oriented requirements analysis phase (Chapter 7).

References

- Allenby, K. and Kelly, T. (2001) 'Deriving Safety Requirements Using Scenarios', in *Proceedings of 5th International Symposium on Requirement Engineering (RE 2001)*, Toronto, Canada, Aug 27-31, pp.228-235.
- Altova-UModel (2014) Commercial UML Tool Trial Version Available at [Online] <http://www.altova.com/umodel.html> [Last Accessed 25 October, 2014]
- Balz, E. and Goll, J. (2005) 'Use Case-Based Fault Tree Analysis of Safety-Related Embedded Systems', in *Proceedings of International Conference on Software Engineering and Applications*, Phoenix, AZ, USA, Nov 14-16, pp.322-330.
- Booch, G., Rumbaugh, J. and Jacobson, I. (2005) *The Unified Modeling Language User Guide*, Addison Wesley, Second Edition.
- Bowles, J. B. and Wan, C., (2001) 'Software Failure Modes and Effects Analysis for a Small Embedded Control System', in *Proceeding of Annual Reliability and Maintainability Symposium (RAMS 2001)*, Philadelphia, Pennsylvania, USA, Jan 22-25, pp.1-6.
- Cepin, M. and Mavko, B. (1999) 'Fault tree developed by an object-oriented method improves requirements specification for safety related systems', *Journal of Reliability Engineering and System Safety*, Vol. 63, No. 2, pp.111-125.
- Cha, S.S., Leveson, N.G. and Shimeall, T.J. (1988) 'Safety Verification in Murphy Using Fault Tree Analysis', in *Proceedings of the 10th International Conference on Software Engineering (ICSE'88)*, Singapore, Apr 11-15, pp. 377-386.
- Chunping, H., Peiqiong, L. and Yiping, Y. (1997) 'The Application of Failure Mode and Effects Analysis for Software in Digital Fly Control System', in *Proceedings of 16th AIAA/IEEE Digital Avionics Systems Conference (DASC)*, Irvine, CA, USA, Oct 26-30, pp.8-13.
- Cichocki, T. and Górski, J. (2000) 'Failure Mode and Effect Analysis for Safety-Critical Systems with Software Components', in *Proceedings of 19th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2000)*, Rotterdam, Netherland, Oct 24-27, pp.382-394.
- Cockburn, A. (2000) *Writing Effective Use Cases*, Addison-Wesley, 2000.

- Cortellessa, V, Singh, H. and Cukic, B. (2002) 'Early reliability assessment of UML based software models', in *Proceeding of 3rd International workshop on Software and performance (WOSP' 02)*, Rome, Italy, Jul 24-26, pp.302-309.
- Cryosat_Rocket_Fault (2005) <http://news.bbc.co.uk/2/hi/science/nature/4381840.stm>(Last Accessed on 20th August, 2014).
- David, P., Idasiak, V. and Kratz, F. (2008) 'Towards a better interaction between design and dependability analysis:FMEA derived from UML/SysML models', in *Proceedings of 17th European Safety and Reliability Conference (ESREL 2008) and SRA-Europe*, Valencia, Spain, Sep 22-25.
- Dehlinger, J. and Lutz, R. R.(2004) 'Software Fault Tree Analysis for Product Lines', in *Proceedings of 8th IEEE International Symposium on High Assurance Systems Engineering (HASE'04)*, Tampa, FL, USA, Mar 25-26, pp.12-21.
- Dehlinger, J. and Lutz, R. R. (2006) 'PLFaultCAT: A Product-Line Software Fault Tree Analysis Tool', *Journal of Automated Software Engineering*, Vol. 13, No. 1, pp.169-193.
- Douglass, B. P. (2009) 'Analyze system safety using UML within Telelogic Rhapsody environment', White Paper, Rational Software, IBM Software Group, April 2009.
- Ern, B, Nguyen, V.Y. and Noll, T. (2013) 'Characterization of Failure Effects on AADL Models', in *Proceedings of 32nd International Conference on Computer Safety, Reliability and Security (SAFECOMP'13)*, Toulouse, France, Sep 24-27, pp. 241-252.
- FaultCAT (2003) Open Source Fault Tree Creation Project [online] <http://www.iu.hio.no/FaultCat> [Last Accessed on 25th October, 2014]
- Federal Aviation Administration (FAA) (2004) Handbook for Object-Oriented Technology in aviation (OOTi) http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot/ [Accessed on April 24, 2014]
- Feng, Q. and Lutz, R. R. (2005) 'Bi-directional safety analysis of product lines', *Journal of Systems and Software*, Vol. 78, No. 2, pp.111-127.
- Friedman, M. A. (1993) 'Automated Software Fault Tree Analysis of Pascal Programs', in *Proceedings of Annual Reliability and Maintainability Symposium (RAMS'93)*, Atlanta, GA, USA, Jan 26-28, pp.458-461.

- Georgieva, K. (2010) 'Conducting FMEA over Software Development Process', *ACM SIGSOFT Software Engineering Notes*, Vol. 35, No. 3, pp.1-5.
- Goddard, P. L. (2000) 'Software FMEA Techniques', in *Proceedings of Annual Reliability and Maintainability Symposium (RAMS 2000)*, LA, USA, Jan 24-27, pp.118-123.
- Gomaa H. (2000) *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison Wesley Object Technology Series, Reading MA, 2000.
- Gorski, J. and Wardzinski, A. (1996) 'Deriving Real-Time Requirements for Software from Safety Analysis', in *Proceedings of 8th Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy, Jun 12-14, pp.9-14.
- Guiochet, J. and Baron, C. (2003) 'UML based FMECA in risk analysis', in *Proceedings of European Simulation and Modeling Conference (ESMc2003)*, Naples, Italy, Oct 27-29, pp.99-106.
- Guichet, J. and Baron, C. (2004) 'UML based risk analysis - Application to a medical robot', in *Proceedings of 5th International Conference on Quality Reliability and Maintenance (QRM'04)*, Oxford, UK, Apr 1-2, pp.213-216.
- Gupta, S., Vinayak, G. V. and Gupta, A. (2012) 'Software Failure Analysis in Requirement Phase', in *Proceedings of 5th India Software Engineering Conference (ISEC 2012)*, Kanpur, UP, India, Feb 22-25, pp.101-104.
- Haapanen, P., and Helminen, A. (2002) 'Failure Mode and Effects Analysis of Software-Based Automation Systems', STUK-YTO-TR, Aug 2002.
- Hansen, K.M., Ravn, A.P. and Starvidou, V. (1998) 'From Safety Analysis to Software Requirements', *IEEE Transactions on Software Engineering*, Vol. 24, No. 7, pp.573-584.
- Harel, D. et al.(1990) 'STATEMATE: A Working Environment for the Development of Complex Reactive Systems', *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, pp.403-414.
- Harvey, P. (1982) 'Fault-tree analysis of software', Master's Thesis, University of California, Irvine, Jan. 1982.

- Hassan, A., Goseva-Popstojanova, K. and Ammar, H. (2005) 'UML Based Severity Analysis Methodology', in *Proceedings of Annual Reliability and Maintainability Symposium (RAMS'05)*, Alexandria, VA USA, Jan 24-27, pp.158-164.
- Hawkins, R. D. (2006) 'Using Safety Contracts in the Development of Safety-Critical Object-Oriented Systems' PhD Thesis, University of York, York, United Kingdom.
- Hawkins, R. D. and McDermid, J. A. (2002) 'Performing hazard and safety analysis of object oriented systems', in *Proceedings of 20th International System Safety Conference (ISSC 2002)*, Denver, CO, USA, Aug 5-9.
- Hecht, H., and Hecht, M. (2004) 'Computer Aided Software FMEA for Unified Modeling Language Based Software', in *Proceedings of Annual Reliability and Maintainability Symposium (RAMS'04)*, LA, USA, Jan 26-29, pp.243-248.
- Heimdahl, M.P. and Leveson, N.G. (1996) 'Completeness and Consistency checking of software requirements', *IEEE Transactions on Software Engineering*, Vol. 22, No. 6, pp.363-377.
- Helmer, G., Wong, J., Slagell, M., Honavar, V., Miller, L. and Lutz, R. R. (2002) 'A Software Fault Tree Approach to Requirements Analysis of an Intrusion Detection System', *Journal of Requirement Engineering*, Vol. 7, No. 4, pp.207-220.
- IEEE-STD-729-1991 (1991) ANSI/IEEE Standard Glossary of Software Engineering Terminology
- Jacobson, I., Christerson, M., Jonsson, P. and Oevergaard, G. (1992) *Object Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.
- Johannessen, P., Grante, C. and Torin, J. (2001) 'Hazard Analysis in Object Oriented Design of Dependable Systems', in *Proceedings of International Conference on Dependable Systems and Networks (DSN 2001)*, Goteborg, Sweden, Jul 1-4, pp. 507-512.
- Kim, H., Wong, W.E., Debroy, V. and Bae, D. (2010) 'Bridging the Gap Between Fault Trees and UML State Machine Diagrams for Safety Analysis', in *Proceedings of 17th Asia Pacific Software Engineering Conference (APSEC'10)*, Sydney, NSW, Australia, Nov 30 – Dec 03, pp. 196-205.

- Kong, W., Shi, Y. and Smidts, C.S. (2007) 'Early Software Reliability Prediction Using Cause-effect Graphing Analysis', in *Proceedings of Annual Reliability and Maintainability Symposium (RAMS 2007)*, Orlando, FL, USA, Jan. 22-25, pp. 173-178.
- Kundu, D. and Samanta, D. (2007) 'An Approach for Assessment of Reliability of the System Using Use Case Model', in *Proceedings of 10th International conference on Information Technology (ICIT 2007)*, Orissa, India, Dec 17-20, pp.243-245.
- Lann, GE (1997) 'An Analysis of the Ariane 5 Flight 501 Failure - A System Engineering Perspective' in *Proceedings of the International Conference and Workshop on Engineering of Computer-Based Systems*, Monterey, CA, USA, Mar 24-28, pp. 339 - 346.
- Lauer, C. and German, R. (2011) 'Fault Tree Synthesis from UML Models for Reliability Analysis at early Design Stage', *ACM SIGSOFT Software Engineering Notes*, Vol. 36, No. 1, pp.1-8.
- Lauritsen, T. and Stalhane, T. (2005) 'Safety Methods in Software Process Improvement', in *Proceedings of 12th European Conference on Software Process Improvement, EuroSPI 2005*, Budapest, Hungary, Nov 9-11, pp.95-105.
- Leveson, N. G. and Harvey, P.R. (1983a) 'Software Fault Tree Analysis', *Journal of Systems and Software*, Vol. 3, No. 2, pp.173-181.
- Leveson, N. G. and Harvey, P.R (1983b) 'Analyzing Software Safety', *IEEE Transactions on Software Engineering*, Vol. 9, No. 5, pp.569-579.
- Leveson, N. G. and Stolzy, J. L. (1983) 'Safety Analysis of Ada Programs Using Fault Trees', *IEEE Transactions on Reliability*, Vol. R-32, No. 5, pp.479-484.
- Leveson, N. G. (1984) 'Software Safety in Computer-Controlled Systems', *IEEE Transactions on Computers*, Vol. 17, No. 2, pp.48-55.
- Leveson, N. G. (1986) 'Software Safety: Why, What and How', *ACM Computing Surveys*, Vol. 18, No. 2, pp.125-163.
- Leveson, N.G., Cha, S.S. and Shimeall, T.J. (1991) 'Safety verification of Ada programs using software fault trees', *IEEE Transactions on Software*, Vol. 8, No.4, pp.48-59.

- Leveson, N.G., Heimdahl, M.P., Hildreth, H. and Reese, J.D. (1994) 'Requirements specification of process-control systems' *IEEE Transactions on Software Engineering*, Vol. 20, No. 9, pp.48-59
- Liu, J., Dehlinger, J. and Lutz, R.R. (2007) 'Safety Analysis of Software Product Lines Using State-Based Modeling', *Journal of Systems and Software*, Vol. 80, No. 11, pp.1879-1892.
- Lu, S., Halang, W. A., Schmidt, H.W. and Gumzej, R. (2005) 'A Component-based Approach to Specify Hazards in the Design of Safety-Critical Systems', in *Proceedings of 3rd IEEE International Conference on Industrial Informatics (INDIN'05)*, Perth, Australia, Aug 10-12, pp.680-685.
- Lu, S., Halang, W. A. and Zaleski, J. (2005) 'Component-based HazOp and Fault Tree Analysis in Developing Embedded Real-Time Systems with UML', in *Proceedings of 4th WSEAS International Conference on ELECTRONICS, CONTROL and SIGNAL PROCESSING*, Miami, Florida, USA, Nov 17-19, pp.150-155.
- Lutz, R. R. and Woodhouse, R. M. (1996) 'Experience Report: Contributions of SFMEA to Requirement Analysis', in *Proceedings of 2nd International Conference on Requirements Engineering (ICRE'96)*, Colorado Springs, Colorado, USA, Apr 15-18, pp.44-51.
- Lutz, R.R. and Woodhouse, R. M. (1997) 'Requirements analysis using forward and backward search', *Annals of Software Engineering*, Vol. 3, No. 1, pp.459-475.
- Lutz, R.R. (1998) 'Safety Analysis of Requirements for a Product Family', in *Proceedings of 3rd International Conference on Requirements Engineering (ICRE'98)*, Colorado Springs, CO, USA, Apr 6-10, pp.24-31.
- Lutz, R., Patterson-Hine, A., Nelson, S., Frost, C. R., Tal, D. and Harris, R. (2007) 'Using obstacle analysis to identify contingency requirements on an unpiloted aerial vehicle', *Journal of Requirements Engineering*, Vol. 12, No. 1, pp.41-54.
- Medikonda BS, Ramaiah PS (2010)'Integrated Safety Analysis of Software-Controlled Critical Systems', *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 1, pp 1-7.

- Meeson, R. (1996) 'Object-oriented-no panacea for safety', in Proceedings of 11th Annual Conference on Systems Integrity, Software Safety and Process Security (COMPASS'96), Gaithersburg, MD, USA, Jun 17-21, pp.171-175.
- Melhart, B. E. (1990) 'Specification and Analysis of the Requirements for Embedded Software with an External Interaction Model', PhD Thesis, University of California, Irvine, 1990.
- Melhart, B. E. (1995) 'Software Fault Tree Analysis for a Requirement System Model', in *Proceedings of Intl. Symposium and Workshop on Systems Engineering of Computer Based Systems*, Tucson, AZ, USA, Mar 6-9, pp.133-140.
- Meng-Lai, Y. Hyde, C. L. and James, L.E. (2000) 'A petri-net approach for early-stage system-level software reliability estimation' In Proceedings of Annual Reliability and Maintainability Symposium (RAMS' 2000), Los Angeles, CA, USA, Jan 24-27, pp.100-105.
- MIL-STD-1629A (1980) 'Procedures for Performing A Failure Mode, Effects and Criticality Analysis', Military Standard, Department of Defense, USA.
- Mohanta, S., Vinod, G., Ghosh, A.K. and Mall, R. (2010) 'An Approach for early Prediction of Software Reliability', *ACM SIGSOFT Software Engineering Notes*, Vol. 35, No. 6, pp.1-9.
- Mojdehbakhsh, R., Subramanian, S., Vishnuvajjala, R., Tsai, W. and Elliott, L. (1994) 'A Process for Software Requirements Safety Analysis', in *Proceedings of 5th Intl. Symposium on Software Reliability Engineering (ISSRE'94)*, CA, USA; Nov 6-9, pp.45-54.
- Musa, J. D. (2002) 'Operational Profiles in Software-Reliability Engineering', *IEEE Transactions on Software*, Vol. 10, No. 2, pp. 14-32.
- Nggada, S. H. (2012) 'Software Failure Analysis at Architecture Level Using FMEA' *International Journal of Software Engineering and Its Applications*, Vol.6, No.1, pp.61-74.
- NASA-GB-8719.13 (2004) NASA Software Safety Guidebook, NASA Technical Standard [<http://www.hq.nasa.gov/office/codeq/doctree/871913.pdf>]
- NASA-STD-8719.13C (2013) NASA Software Safety Standard, NASA Technical Standard [<http://www.hq.nasa.gov/office/codeq/doctree/NS871913C.pdf>]

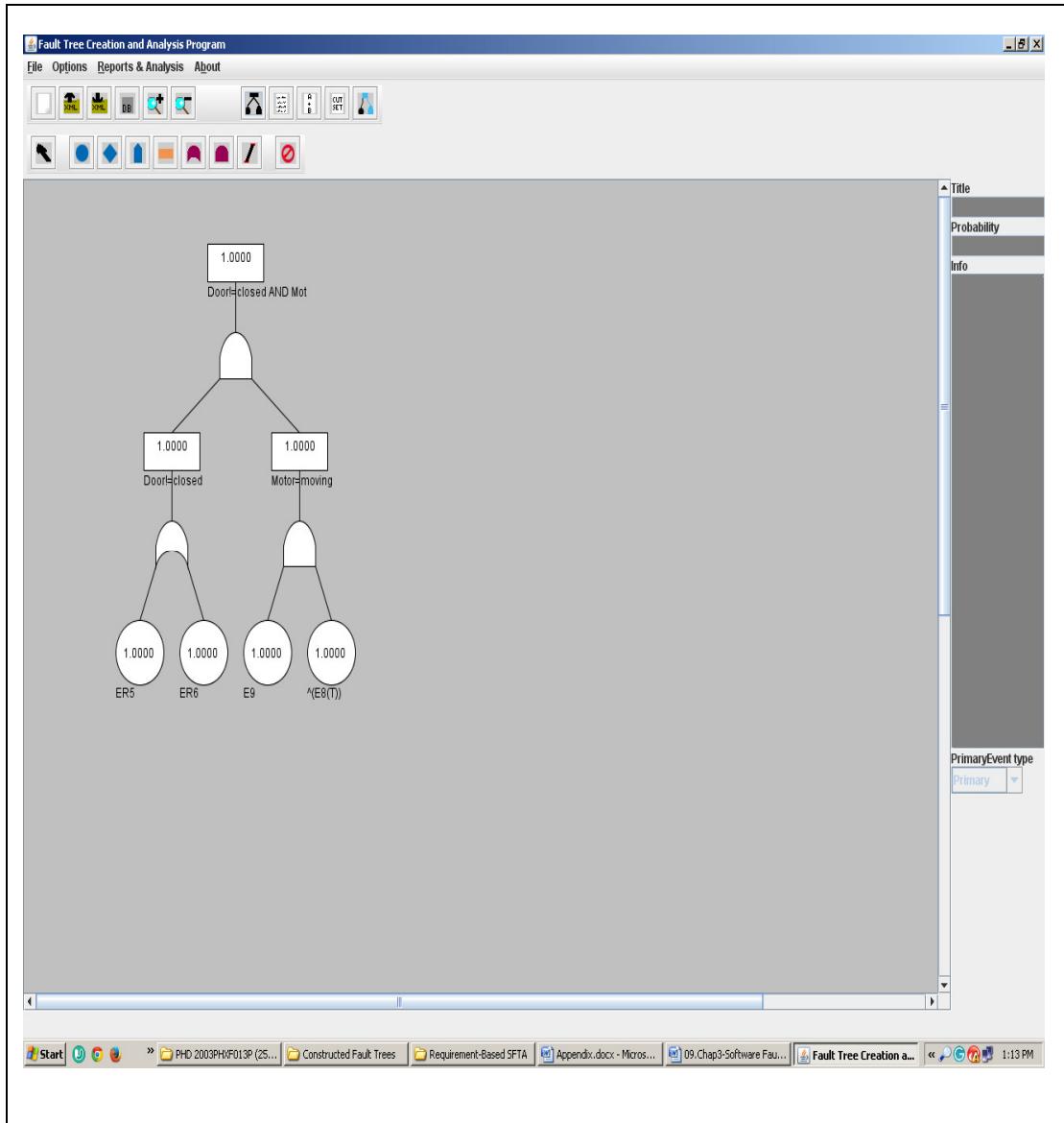
- Ordonio, R.R. (1993) 'An Automated Tool to Facilitate Code Translation for Software Fault Tree Analysis', M.S. Thesis, Naval Postgraduate School, Monterey, CA, USA.
- Ozarin, N. and Siracusa, M. (2003) 'A Process for Failure Modes and Effects Analysis of Computer Software', in *Proceedings of Annual Reliability and Maintainability Symposium (RAMS'03)*, Tampa, Florida, USA, Jan 27-30, pp.365-370.
- Ozarin, N. (2004) 'Failure Mode and Effects Analysis during Design of Computer Software', in *Proceedings of Annual Reliability and Maintainability Symposium (RAMS'04)*, LA, USA, Jan 26-29, pp.201-206.
- Pai, G. J. and Dugan, J.B. (2002) 'Automatic Synthesis of Dynamic Fault Trees from UML System Model', in *Proceedings of 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, Maryland, USA, Nov 12-15, pp.243-256.
- Price, C. and Snooke, N. (2008) 'An Automated Software FMEA', in *Proceedings of International System Safety Regional Conference (ISSRC 2008)*, Singapore, Apr 23-25.
- Qantas_Flight_72 (2008) http://en.wikipedia.org/wiki/Qantas_Flight_72 (Last Accessed on 20th August, 2014)
- Ratan, V., Partridge, K., Reese, J. and Leveson, N. (1996) 'Safety Analysis Tools for Requirements Specifications', in *Proceedings of the 11th Annual Conference on Computer Assurance Systems Integrity Software Safety Process Security (COMPASS '96)*, Gaithersburg, MD, USA, Jun 17-21, pp.149-160.
- Reifer, D. J. (1979) 'Software Failure Modes and Effects Analysis', *IEEE Transactions on Reliability*, Vol. R-28, No. 3, pp.247-249.
- Reid, W.S. (1994) 'Software Fault Tree Analysis of Concurrent Ada Processes', MS Thesis, Naval Postgraduate School, Monterey, California, USA.
- Reinhardt, D. W. (2004) 'Use of the C++ Programming Language in Safety Critical Systems', Master Thesis, University of York, USA.
- Schellhorn, G., Thums, A. and Reif, W. (2002) 'Formal Fault Tree Semantics', in *Proceedings of the 6th World Conference on Integrated Design and Process Technology (IDPT' 2002)*, Pasadena, CA, USA, Jun 23-28.

- Singh, H., Cortellessa, V, Cukic, B, Gunel, E. and Bharadwaj, V. (2001) 'A bayesian approach to reliability prediction and assessment of component based system', in *Proceedings of 12th International Symposium on Software Reliability Engineering (ISSRE'92)*, Hong Kong, China, Nov. 27-30, pp.12-21.
- Snooke, N. (2004) 'Model-based Failure Modes and Effects Analysis of Software', in *Proceedings of 15th International Workshop on Principles of Diagnosis (DX-2004)*, Carcassonne, France, Jun 23-25, pp.221-226.
- Snooke, N. and Price, C. (2011) 'Model-driven Automated Software FMEA', in *Proceedings of Annual Reliability and Maintainability Symposium (RAMS'11)*, Lake Buena Vista, FL, USA, Jan 24-27, pp.1-6.
- Sommerville, I. (2005) *Software Engineering*, Seventh Edition, Pearson Addison Wesley, 2005.
- Stadler, J.J and Seidl, N.J. (2013) 'Software Failure Modes and Effects Analysis', in *Proceedings of Annual Reliability and Maintainability Symposium (RAMS'13)*, Orlando,FL,USA, Jan 28-31, pp. 201-206.
- Taylor, J. R. (1982) 'Fault Tree and Cause Consequence Analysis for Control Software Validation', Technical Report, RISO-M-2326, Risd National Laboratory, Denmark.
- Tiwari, S., Rathore, S. S., Gupta, S., Gogate, V. and Gupta, A. (2012) 'Analysis of Use Case Requirements Using SFTA and SFMEA Techniques', in *Proceedings of 17th International Conference on Engineering of complex Computer Systems (ICECCS, 2012)*, Paris, France, Jul 18-20, pp. 29-38.
- Towhidnejad, M, Wallace, D. R. and Gallo, A. M. (2002) 'Fault Tree Analysis for Software Design', in *Proceedings of 27th Annual NASA Goddard/IEEE Software Engineering Workshop (SEW'02)*, Greenbelt, Maryland, USA, Dec 5-6, pp.24-30.
- Towhidnejad, M., Wallace, D.R. and Gallo, A. M. (2003) 'Validation of Object Oriented Software Design with Fault Tree Analysis', in *Proceedings of 28th Annual NASA Goddard/IEEE Software Engineering Workshop (SEW'03)*, Greenbelt, Maryland, USA, Dec 3-4, pp.209-215.

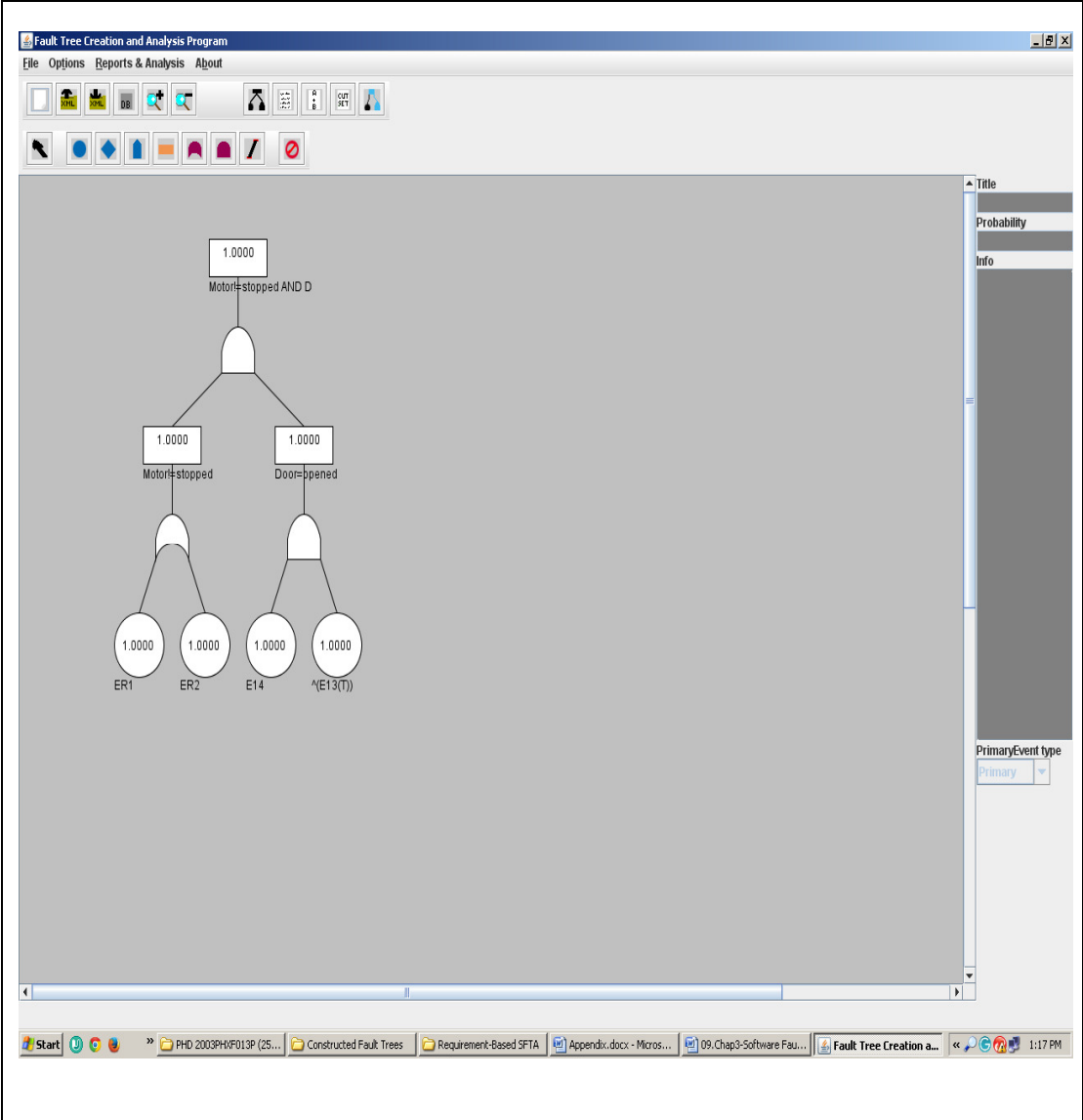
- Tripathi, R. and Mall, R. (2005) 'Early Stage Software Reliability and Design Assessment', in *Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, Taipei, Taiwan, Dec 15-17, pp.341-348.
- Troubitsyna, E. (2011) 'Failure Modes and Effects Analysis of Use Cases: A Structured Approach to Engineering Fault Tolerant Requirements', in *Proceedings of 4th International Conference of Dependability (DEPEND 2011)*, French Riviera, Nice/Saint Laurent du Var, France, Aug 21-27, pp.82-87.
- Tsuchiya, T., Terada, H., Kusumoto, S., Kikuno, T. and Kim, E. M. (1997) 'Derivation of Safety Requirements for Safety Analysis of Object-Oriented Design Documents', in *Proceedings of 21st Annual International Conference on Computer Software and Applications (COMPSAC 1997)*, Washington, DC, USA, Aug 11-15, pp.252-255.
- Vesely, W.E., Goldberg, F.F., Roberts, N.H. and Haasl, D.F. (1981) *Fault Tree Handbook*, NUREG-0492, U.S. Nuclear Regulatory Commission, Washington DC, USA.
- Vyas, P. and Mittal, R. K. (2009) 'Operation Level Safety Analysis for Object Oriented Software Design Using SFMEA', in *Proceedings of International Advance Computing Conference (IACC'2009)*, Patiala, India, Mar 6-7, pp.1675-1679.
- Vyas, P. and Mittal, R. K. (2012) 'Eliciting Additional Safety Requirements from Use Cases using SFTA', in *Proceedings of 1st International Conference on Recent Advances in Information Technology (RAIT 2012)*, Dhanbad, India, Mar 15-17, pp.163-169.
- Vyas, P. and Mittal, R. K. (2013) 'Hazard analysis of Unified Modelling Language sequence and state charts using software fault tree analysis', *International Journal of Critical Computer-Based Systems (IJCCBS)*, Vol. 4, No. 2, pp.173-197.
- Vyas, P. and Mittal, R. K. (2015) 'The applications of SFTA and SFMEA approaches during software development process: an analytical review', *International Journal of Critical Computer-Based Systems (IJCCBS)*, Vol. 6, No. 1, pp. 29-49.

- Wentao, W. and Hong, Z. (2009) 'FMEA for UML-based Software', *World Congress on Software Engineering (WCSE'09)*, Xianmen, China, May 19-21, pp.456-460.
- Weber, W., Tondok, H. and Bachmayer, M. (2005) 'Enhancing software safety by fault trees: experiences from an application to flight critical software', *Journal of Reliability Engineering and System Safety*, Vol. 89, No. 1, pp.57-70.
- Winter, M. W. and Shimeall, T. J. (1995) 'Software Fault Tree Analysis of an Automated Control System Device Written in Ada', M.S Thesis, Naval Postgraduate School, Monterey, California.
- Yacoub, S, Cukic, B. and Ammar, H. H. (2004) 'A scenario-based reliability analysis approach for component-based software', *IEEE Transactions on Reliability*, Vol.53, No. 4, pp. 465-480.

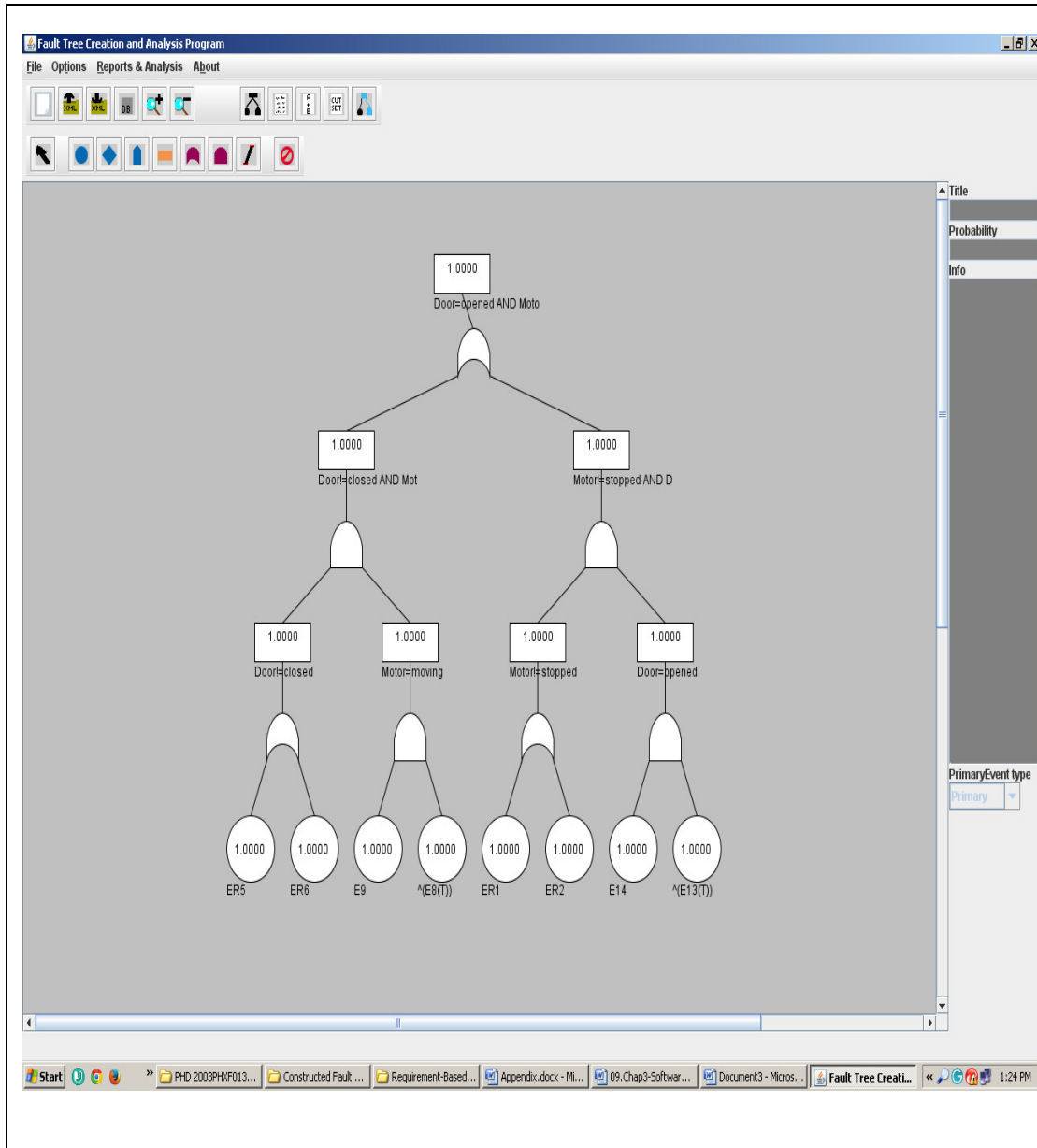
Fault Tree Constructed from the faulttree.xml file of Figure 3.14 Using FaultCAT Tool



Fault Tree Constructed from the faulttree.xml file of Figure 3.16 Using FaultCAT Tool

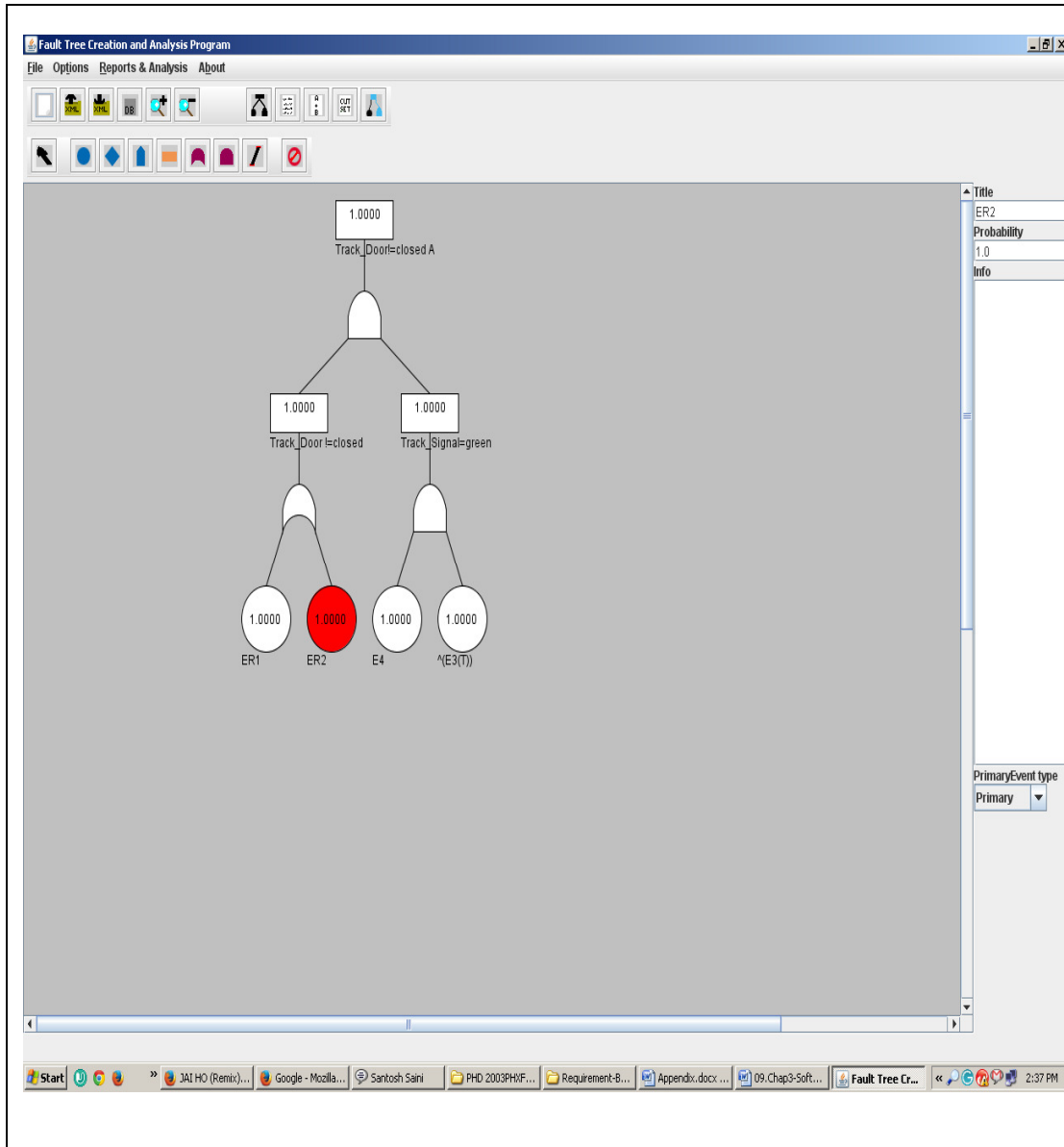


Fault Tree Constructed from the faulttree.xml file of Figure 3.18 Using FaultCAT Tool

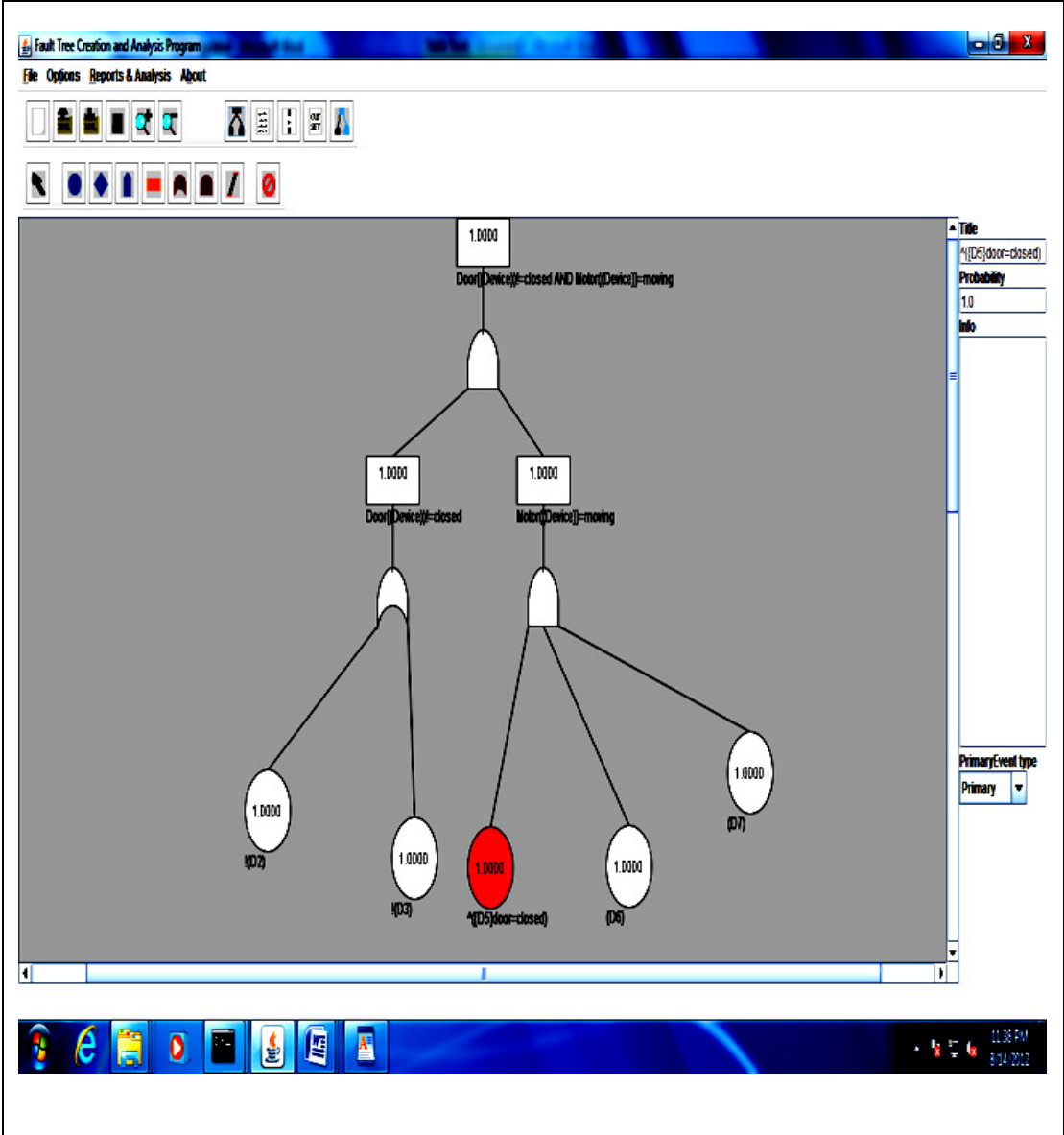


Appendix-IV

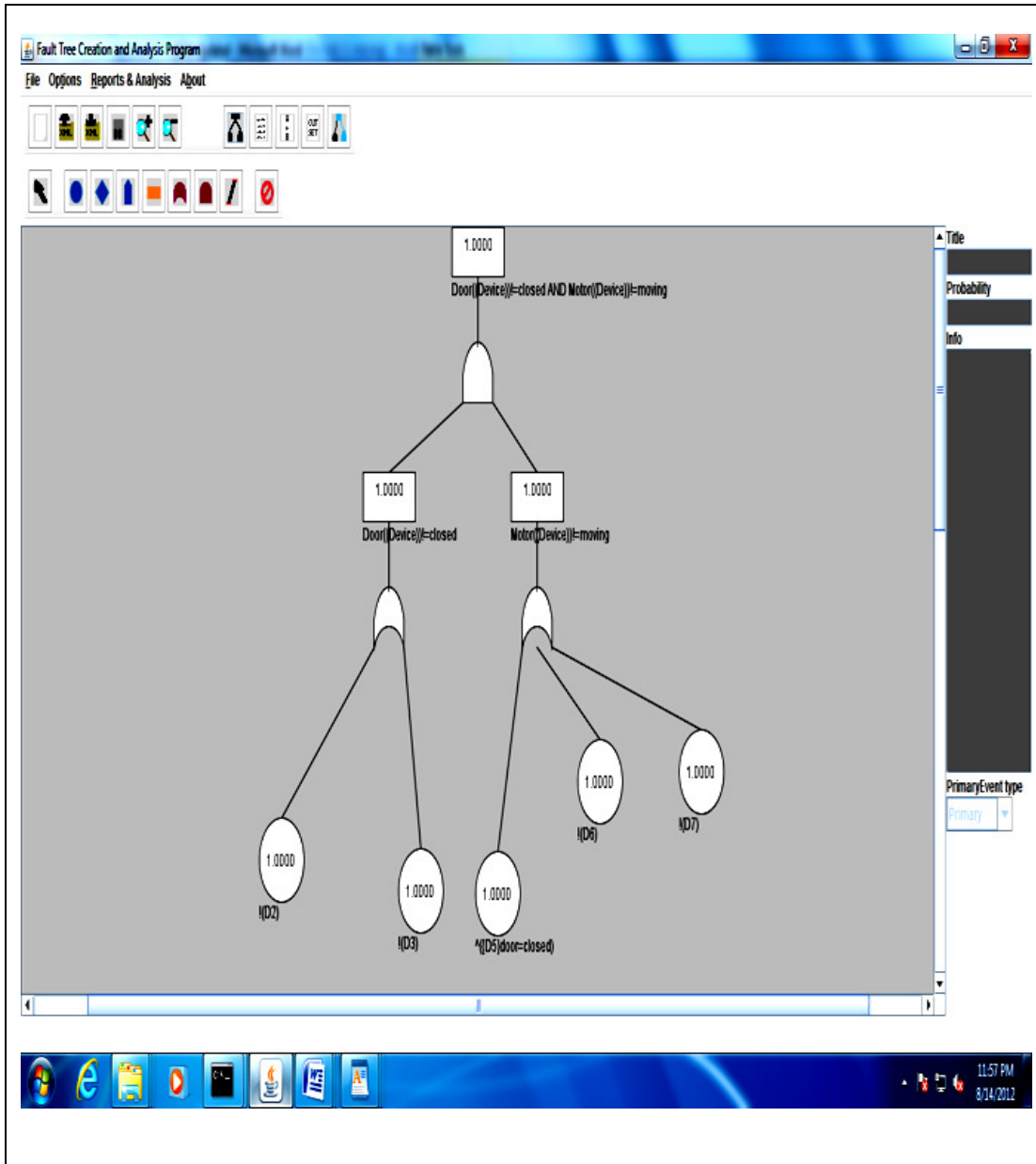
Fault Tree Constructed from the faulttree.xml file of Figure 3.22 Using FaultCAT Tool



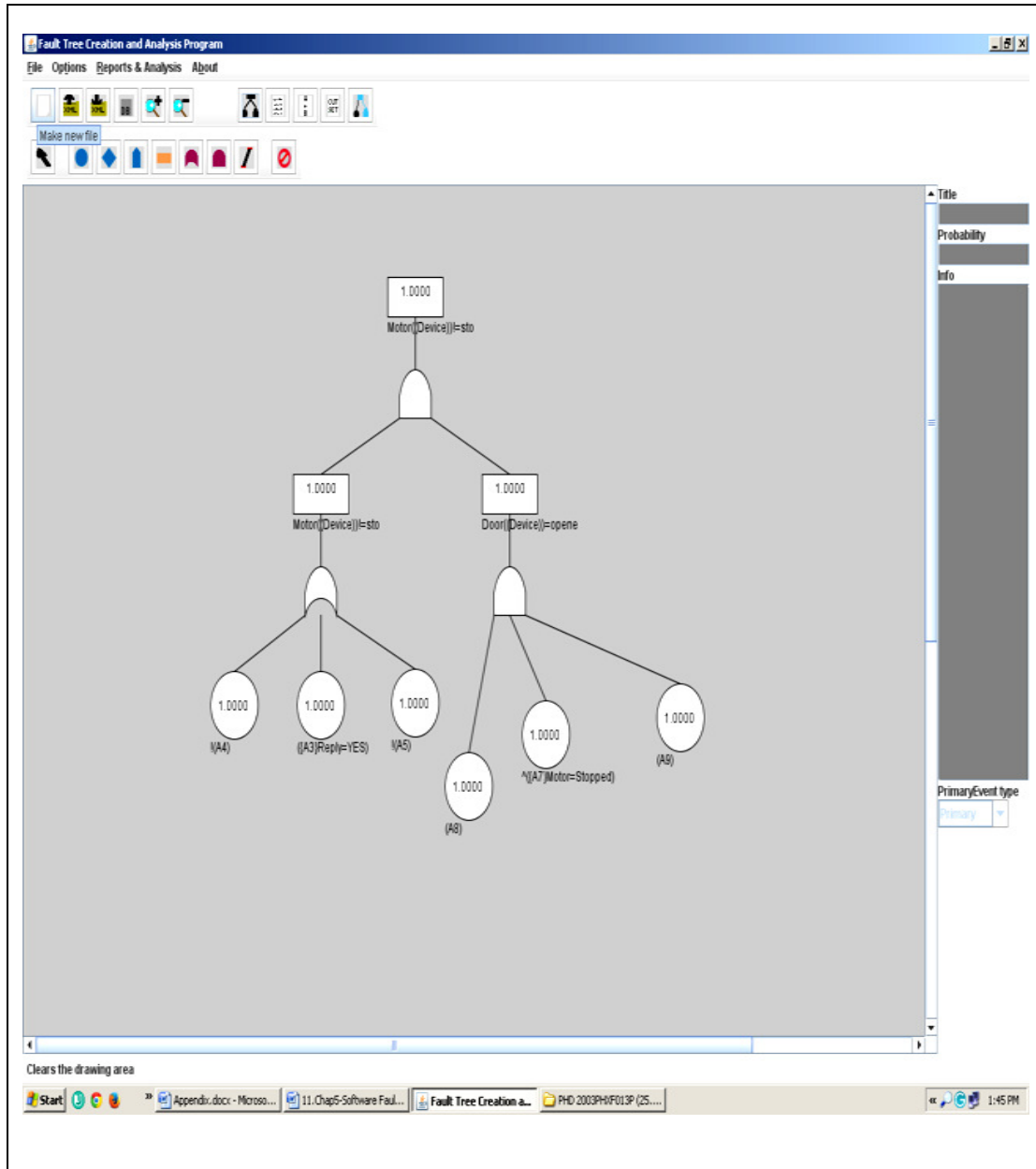
Fault Tree Constructed from the faulttree.xml file of Figure 5.5 Using FaultCAT Tool



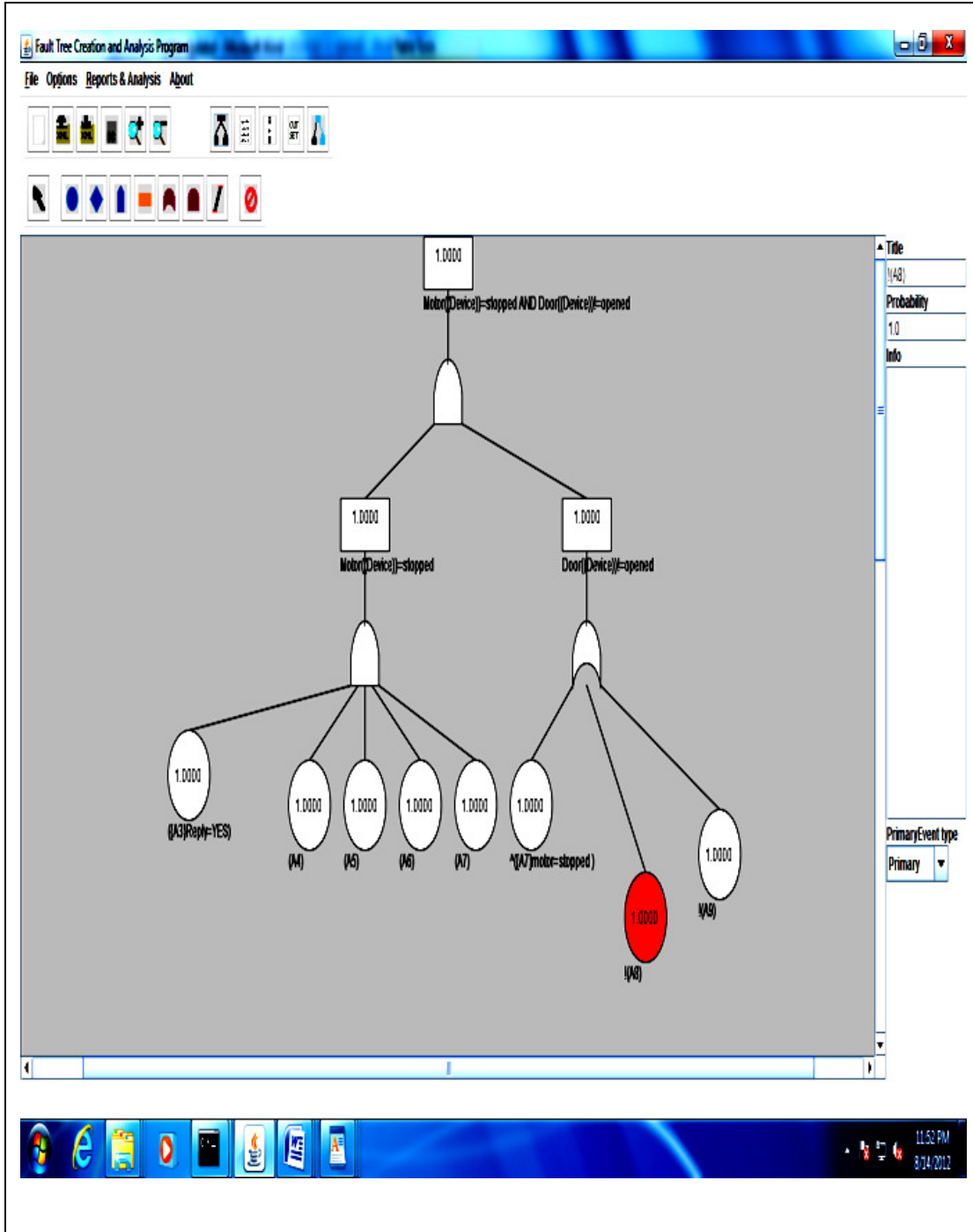
Fault Tree Constructed from the faulttree.xml file of Figure 5.7 Using FaultCAT Tool



Fault Tree Constructed from the faulttree.xml file of Figure 5.11 Using FaultCAT Tool



Fault Tree Constructed from the faulttree.xml file of Figure 5.13 Using FaultCAT Tool



List of Publications

JOURNALS

- [1] Vyas, P. and Mittal, R.K. (2013) ‘Hazard analysis of Unified Modelling Language sequence and state charts using software fault tree analysis’, *International. Journal of Critical Computer-Based Systems (IJCCBS)*, Vol.4, No.2, pp.173-197.
- [2] Vyas, P. and Mittal, R.K. (2015) “The Applications of SFTA and SFMEA Approaches During Software Development Process: An Analytical Review”, *International. Journal of Critical Computer-Based Systems (IJCCBS)*, Vol.6, No.1, pp.29-48.

CONFERENCES

- [1] Vyas, P. and Mittal, R.K. (2006) ‘Application of Safety Analysis Techniques in Object Oriented in Software Design’, in *3rd International Conference On Quality, Reliability AND Infocomm Technology (ICQRIT’ 2006)*, New Delhi, Dec 2-4.
- [2] Vyas, P. and Mittal, R.K. (2009) ‘Operation Level Safety Analysis for Object Oriented Software Design Using SFMEA”, in *Proceedings of International Advance Computing Conference (IACC’2009)*, Patiala, India, Mar 6-7, pp.1675-1679.
- [3] Vyas, P. and Mittal, R.K. (2012) “Eliciting Additional Safety Requirements from Use Cases using SFTA”, in *Proceedings International Conference on Recent Advances in Information Technology(RAIT 2012)*, Dhanbad, India, Mar 15-17, pp.163-169.

COMMUNICATED

- [1] Vyas, P. and Mittal, R.K. (2014) “Automated SFTA Approach for Use-case based Requirements Analysis Process”, *International Journal of Automated SoftwareEngineering*, Springer-Verlag
- [2] Vyas, P. and Mittal, R.K. (2014) “A Novel Approach for Early Prediction of Software Reliability of Use-Cases”, *International Journal of Reliability and Safety (IJRE)*, Inderscience Publishers.
- [3] Vyas, P. and Mittal, R.K. (2014) “Safety Analysis of UML Dynamic Models Using Automated Software FMEA Approach”, in *Proceedings of 8th India Software Engineering Conference (ISEC 2015)*.

Brief Biography of the Candidate

PANKAJ VYAS received the B.E. (Computer Science) degree from Amravati University, Amravati (INDIA) in 1993 and the M.E. degree in Computer Science from Birla Institute of Technology and Science (BITS), Pilani (INDIA) in 2002. He had taught at SRPA AB College Pathankot for eight years from 1994 to 2002. Since 2002, he is working as a Lecturer in the Department of Computer Science and Information System at BITS-Pilani, Pilani Campus and since then he has taught several courses namely 'Object Oriented Programming', 'Advanced Operating Systems', 'Software Engineering' etc. His research interests include software quality improvement, safety enhancement of object-oriented systems, software safety analysis techniques and early software reliability estimation of object-oriented software systems.

Brief Biography of the Supervisor

R K MITTAL is currently Senior Professor and Director (Special Projects) at Birla Institute of Technology & Science, Pilani. Before that he was a Director of BITS-Pilani, Dubai Campus, Dubai from July 2010 to March 2014, where he introduced several reforms and brought the changes to align with BITS, Pilani.

He obtained his B.E, M.E and Ph.D Degrees from BITS-Pilani, Pilani Campus. He was awarded Institute's Gold Medal for standing First in M.E. program. Prof. Mittal was leading the Academic Registration and Counselling Division as the Dean until he was appointed as Deputy Director Administration of BITS Pilani, Pilani Campus in 2009. He is credited and acknowledged for computerizing the student registration process and student records, accounts and other processes of the university and spearheading the BITS Alumni Association (BITSAA). He was also instrumental in establishing and developing the state-of-the-art Centre for Robotics at BITS-Pilani. Under his leadership, the institute received achieved the rare distinction of developing the first ever Indian prototype of a humanoid, "ACYUT", widely acclaimed in several competitions and the press alike. He is the co-author of two bestselling books, "Elements of Manufacturing Processes" (PHI) and "Robotics & Control" (TMH). His research interests include software reliability, MEMS/NEMS, nanotechnology, robotics, robust robot motion planning, vision based robot guidance, mechatronics, modelling and environmental policy.

He has published extensively in various peer-reviewed, SCI indexed and high impact national and international journals. Under his able guidance, the Mechanical Engineering department conducted its first international conference in 2007 "Emerging Mechanical Technologies: Macro to Nano" (EMTM2N-2007). He was also instrumental in the successful conduct of the first International Conference on Cloud Computing in the Middle East region ICCTAM-12 at BITS-Pilani, Dubai Campus. He has delivered keynotes, invited talks and chaired numerous sessions in International Conferences in India and abroad. He has guided four Ph.D. students and is currently guiding four Ph.D. students.