University of Denver
Digital Commons @ DU

Electronic Theses and Dissertations | Graduate Studies

11-1-2014

Fail-Safe Testing of Safety-Critical Systems

Ahmed Gario
University of Denver

Follow this and additional works at: https://digitalcommons.du.edu/etd

 Part of the Computer Sciences Commons

Recommended Citation

Gario, Ahmed, "Fail-Safe Testing of Safety-Critical Systems" (2014). *Electronic Theses and Dissertations*. 230.
https://digitalcommons.du.edu/etd/230

This Dissertation is brought to you for free and open access by the Graduate Studies at Digital Commons @ DU. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of Digital Commons @ DU. For more information, please contact jennifer.cox@du.edu,dig-commons@du.edu.

Fail-Safe Testing of Safety-Critical Systems

A Dissertation

Presented to the Faculty

of the Daniel Felix Ritchie School of Engineering and Computer Science

University of Denver

in Partial Fulfillment

of the Requirements for the Degree

of Doctor of Philosophy

by

Ahmed Gario

November 2014

Advisor: Anneliese Andrews

© Copyright by Ahmed Gario, 2014

All Rights Reserved

Author: Ahmed Gario
Title: Fail-Safe Testing of Safety-Critical Systems
Advisor: Anneliese Andrews
Degree Date: November 2014

Abstract

This dissertation proposes an approach for testing of safety-critical systems. It is based on a behavioral and a fault model. The two models are analyzed for compatibility and necessary changes are identified to make them compatible. Then transformation rules are used to transform the fault model into the same model type as the behavioral model. Integration rules define how to combine them. This approach results in an integrated model which then can be used to generate tests using a variety of testing criteria. The dissertation illustrates this general framework using a CEFSM for the behavioral model and a Fault Tree for the fault model. We apply the technique to a variety of applications such as a Gas burner, an Aerospace Launch System, and a Railroad Crossing Control System. We also investigate the scalability of the approach and compare its efficiency with integrating a state chart and a fault tree. Construction and Analysis of Distributed Processes (CADP) has been used as a supporting tool for this approach to generate test cases from the integrated model and to analyze the integrated model for some properties such as deadlock and livelock.

Acknowledgements

I would like to express my deep gratitude to my adviser, Dr. Anneliese Andrews for her unlimited precious guidance, motivation and direction throughout my research work and in preparation for this dissertation.

I would like to thank Dr. Gareth Eaton (Department of Chemistry & Biochemistry) for accepting my request to be the examining committee chair and for his valuable comments. I would like also to thank the examining committee members: Dr. Matthew Rutherford and Dr. Rinku Dewri for their participation in the committee and for their excellent comments.

I would like to thank my friends and colleagues whom I met and worked with during my PhD journey. Special thanks for Salwa Elakeili whom I worked with on case studies and on the end-to-end testing methodology. I would also like to thank Mahmoud Abdelgawad with whom I worked closely on modeling the environment and with whom I exchanged ideas. It was a pleasure to work with Seana Hagerman on some publications and I thank her for providing me with the description for the launch vehicle system which I used as a case study. I would like also to thank Andrei Roudik, our system administrator at the Department of Computer Science, for providing the technical support whenever we asked for it. I would like to thank NSF/SSR-RC for supporting this dissertation.

My thoughts and deepest gratitude go straight to my mother, sisters, and brothers for their love, encouragement, and support throughout this journey.

Last but not least, I would like to express the most profound gratitude for my wife for being patient, supportive, and amiable no matter what. No gratitude would be enough for my children, Yousef, Dania, Omar, and Randa who tolerated my student life in which they wanted me when I was not there.

Contents

List of Tables

List of Figures

Chapter 1

Problem Statement

1.1 Introduction

Systems, especially those that rely on software, have become an essential part of our world. From an engineering point of view, software systems are ubiquitous. These systems, where human safety depends upon their correct operation, are considered safety-critical and are part of our daily life. An obvious example of a safety critical system is an aircraft fly-by-wire control system. In this system the pilot uses an interface to input commands to the control computer, and the computer controls the actual aircraft. The lives of hundreds of passengers depend upon the continued correct operation of such a system.

Railway signaling systems must enable controllers to direct trains and prevent trains from colliding. Like an aircraft fly-by-wire, lives depend on the correct operation of the system. However, all trains can be stopped if the safety of the system becomes suspect, while stopping an aircraft to fix the fly-by-wire system when flying is not possible. Software in medical systems may be directly responsible for human life, such as information which a doctor uses to decide on medication [18].

Software may also be involved in providing humans with information, such as a pacemaker device. Both types of systems can impact the safety of the patient. A pacemaker device is a safety-critical system, since its failure may cause severe damage to a human body or even loss of a human life. It is an electronic device used to treat patients who suffer from slow heartbeats. The purpose of using a pacemaker is to maintain normal heartbeats so that adequate oxygen and nutrients can be delivered through the blood to the organs of the human body.

Civil engineers use computer software to design and test structure models. An error in the software may result in a bridge collapsing. Aircraft, trains, ships and cars are also designed and modeled using computers. Even something as simple as traffic lights or a microwave oven can be viewed as safety-critical. An error giving green lights to both directions at a cross road could result in a car accident, or a microwave sending out waves while the oven door is open could result in human injury. In cars, software involved in functions such as engine management, anti-lock brakes, traction control, and a host of other functions, could potentially fail in a way that causes a road accident.

With such systems also comes the exposure to risks because some of these systems may fail or may not work properly resulting in damage, injury, or death. Potential system failures that result in damage, injury, or death are referred to as a mishap risk. Therefore, safety must always be considered with respect to all system components (the software and the computer hardware, other electronic and electrical hardware, mechanical hardware, and operators or users) not just the software element.

Hazards will always exist, but their risk must be mitigated. Therefore, safety is a relative term that refers to the level of risk that is acceptable. System safety is not an absolute quantity, but rather a level of risk that is bound by cost, time, and

performance. System safety requires the evaluation of risk to determine its level in order to decide whether to accept or reject it. System safety is achieved through a sequence of ordered steps applied from the initial system design, through detailed design and testing, to the end of a system's lifetime.

1.2 The Cost of Software Safety

The overall cost of software safety is determined by what we are willing to pay. It includes many factors and components, and depends on whether we want to pay to prevent the hazards from happening or we want to pay after the occurrence of hazards. The preventive approach, the cost that we spend to produce safer software that eliminates software defects in general, is only one proportion of the cost to develop safety-critical software that can save human lives and properties. The other one is the cost we pay after the occurrence of an accident or mishap [18]. The problem is we cannot ever be certain that the system operates safely. On the other hand, when accidents do occur, the penalties for ignoring software safety can be very severe. Therefore, a preventive approach for safety during design is more cost-effective than trying to implement safety into a system after the occurrence of an accident or mishap since the cost of mishap could be exorbitant [43].

1.3 Safety Analysis

Safety analysis is the examination and evaluation of the system and subsystem to find and categorize the existing and potential hazards and hazardous conditions according to their severity and frequency of occurrence and to attain the proper measures to mitigate them [43]. It increases the probability of finding possible

faults in safety-critical software. Some safety analysis techniques have been used to analyze safety in safety-critical software. The outcome of this analysis is considered when testing safety in the behavioral model.

Besides the techniques used in testing software generally, testing safety-critical software systems requires analyzing the hazards beforehand by using analysis techniques such as Fault Tree Analysis (FTA) [143], Failure Modes and Effects Analysis (FMEA) [123], Hazard and Operability Analysis (HAZOP) [83], and Hazard and Risk Analysis (HRA) [43, 60]. However, there is still a gap between the testing and analysis activities which negatively impacts the effectiveness of testing. Another difference is that when testing safety-critical software we are testing the undesired behaviors which are not described by the system model, while when testing non-safety critical software we are testing the desired behaviors, i.e. how the system is supposed to behave.

1.4 Software Testing

Software testing is a very important activity in the software development life cycle. It is the process of operating the software under a controlled condition to verify that it behaves as specified, to detect its errors, and to validate that it is what the customer wants. Validation checks to see if we are building the right system and Verification checks to see if we are building the system correctly [4]. However, the purpose of Error Detection is to find out whether things happen when they should't or they don't happen when they should. Both verification and validation are necessary, but are different components of any testing activity. According to the ANSI/IEEE1059 standard, testing is defined as the process of analyzing software to

detect the differences between existing and required conditions and to evaluate the features of the software item.

The main purpose of testing software is to find errors and problems. The point of finding these errors and problems is to have them fixed. Testing in itself cannot ensure the quality of software, all it can do is to give us a certain level of assurance about the software based on given controlled conditions. In other words, testing shows us whether the software functions as expected or not under the test cases executed. This level of assurance depends on the testing criteria and strategy applied, and techniques and tools used. A well-designed test case may reveal previously undetected software defects [145].

In well-organized projects, the mission of the testing team is not only to perform testing, but also to help minimize the risk of product failure. Testers in the testing team do not look only for obvious errors in the product, but also try to find potential problems and the absence of problems [26]. They examine and report the quality of the product based on specific criteria, so that a release decision about the software can be made.

1.5 Testing Problem for Safety-Critical Systems

In safety-critical systems, we are concerned with the desired behavior as much as the undesired behavior. The undesired behavior may cause injuries or loss of life to people, or property damage. Therefore, it is essential to lower the probability of the occurrence of the hazards as well as its severity by implementing mitigation actions for the system. Thus, we need to:

- generate behavioral test to test the behavior of the system,

- generate failures at appropriate points ex. by injecting events into the system model or by manipulating the sensor values, and
- test proper mitigation.

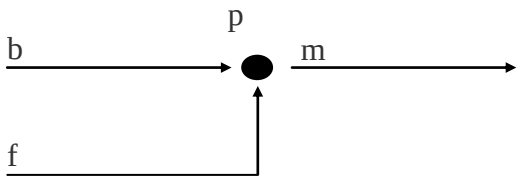

Figure 1.1: Overall Approach

From the test process shown in Figure 1.1, we need to know:

- What behavior (b) the model describes,
- with which failures (f) are we testing, including multiple criteria and priorities,
- at what point (p) in the behavior, and
- with which mitigation model (m).

To overcome the safety-critical system testing problems such as the gap (between the desired and the undesired behaviors) introduced by the separation of models, we need to integrate these models such that they can be used for testing as well as having testing criteria for the this integrated model.

1.6 Model Based Testing (MBT)

Model-based testing (MBT) [31] is common for functional testing. Models provide a functional view of the system that can be used to produce test cases without using the actual system implementation details because it is very difficult to cover

all code structures especially for complex dependable systems [140]. MBT focuses on testing the functional or the expected behavior (also known as desired behavior). Models commonly used in this context include the Unified Modeling Language (UML) [125], Finite State Machines (FSM), Extended Finite State Machines (EFSMs), and Communicating Extended Finite State Machines (CEFSMs). Each type of model has its own characteristics that makes it more suitable for one kind of behavior than for others. For example, CEFSMs are better in modeling communication between processes than FSMs because CEFSMs have the capabilities to handle the communication part whereas FSMs do not.

MBT is intended for testing the desired behavior that the model describes. However, safety is not described by the system models and therefore, it has to be analyzed separately by one of the safety analysis techniques. The output of this analysis can then be used for safety testing. To analyze safety, different models, known as fault models, which describe the undesired behavior, are used. These models are different from the models that describe the desired behavior. Although the output of safety analysis techniques is used for testing, the separation of these models hinders achieving adequate safety testing since the output of the analysis tells what events contribute to the occurrence of the hazard but does not tell how and when the hazard would occur during the system execution. That is due to the fact that these two activities are conducted separately on different models for different goals.

This leads to the idea of model integration. The idea of model integration is to combine fault models with behavioral models in order to know not only what causes hazards in the system, but also when and how these hazards occur during system execution. In this dissertation, we will use CEFSM for model integration due to its capabilities such as the flatness of the model and the explicit interaction power. CEFSM has also been extensively used in modeling communicating and embedded

systems. We propose transformation rules to transform fault trees gate by gate into their equivalent CEFSMs. Then we integrate the CEFSM form of the fault tree with behavioral models according to transformation rules. The output of the integration process is an integrated model. The behavioral model and fault model can be thought of as communicating processes where the fault process will receive events from the behavioral process that may contribute directly to a hazard. The integrated model can be used for safety analysis, safety testing, and testing proper mitigation. The complexity, scalability, effectiveness, and efficiency of the proposed approach are investigated.

1.7 Research Questions

- **RQ1:** Can we combine behavior models with fault models to be used for testing software safety?
- **RQ2:** How can we overcome the limitations such as scalability and complexity that the other approaches introduced?
- **RQ3:** Can we overcome the limitation of compatibility between the behavior and fault model and how?
- **RQ4:** Can we create test cases that target hazards from the integrated model?
- **RQ5:** Can the integrated model be used for safety analysis as well?
- **RQ6:** Can we define test criteria that are suitable for the integrated model and target safety?
- **RQ7:** Can the approach be used within an end-to-end testing methodology?
- **RQ8:** Can we generalize the integration approach so that it fits other BM and FM?
- **RQ9:** Can we validate the approach?

- **RQ10:** Can the approach be applied to all types of safety critical systems?
- **RQ11:** Can we use this approach with tool support?

The rest of this document is organized as follows: Chapter 2 provides a background overview of hazard analysis, hazard testing, Safety-Critical System Lifecycle (SCSL), MBT, and model integration. Our approach which includes fault model transformation and integration rules is described in chapter 3. Chapter 4 validates the approach. It investigates scalability and applicability. It also provides a case study that shows using the approach as part of an end-to-end testing methodology. Chapter 5 discusses other uses of the integrated model using CADP. We conclude in Chapter 6. Finally, chapter 7 explores the future work.

1.8 Contribution to Team Project

This dissertation is a part of a large project to test a proper failure mitigation and security. This project consists of the following:

- A (Ahmed Gario): This part of the project is this dissertation. It focuses on generating failures by integrating fault trees and the behavioral model according to a set of transformation and integration rules.
- B (Salwa Elakeili): This part of the project concentrates on testing the proper mitigation of safety-critical system.
- C (Seana Hagerman): This part uses the attack tree to test security of the proper mitigation.
- D (Mahmoud Abdelgawad): This portion emphasizes on testing autonomous system.

- E: This part of the project interested in generating fail-safe tests for Web application.

Figure 1.2 illustrates how these parts of the overall large project are related, and Table 1.1 describes the overlapping areas in more details.

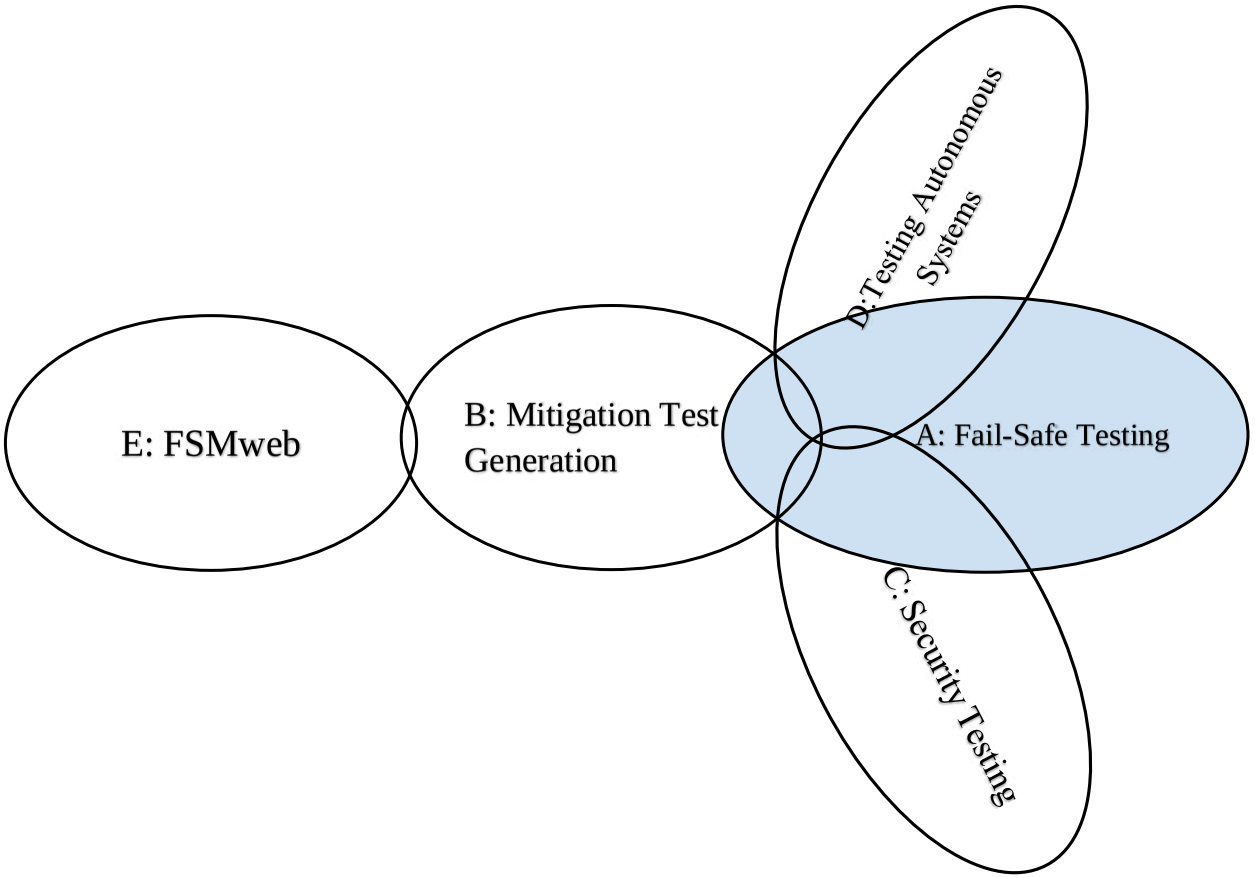

Figure 1.2: Team Contribution

Table 1.1: Project Contribution Commonality Table

	A	B	C	D	E
A	-	End-to-End Approach, Modeling language used (CEFSM), Case Studies	Approach, Case Studies	Modeling language used (CEFSM), Approach	
B	End-to-End Approach, Modeling language used (CEFSM), Case Studies	-	Case Studies, Mitigation	Modeling language used (CEFSM)	Regression testing, Weaving rules, Comparison
C	Approach, Case Studies	Case Studies, Mitigation	-	Modeling language used (CEFSM)	
D	Modeling language used (CEFSM), Approach	Modeling language used (CEFSM)	Modeling language used (CEFSM)	-	
E		Regression testing, Weaving rules, Comparison			-

Chapter 2

Background

This chapter provides an overview of the Safety-Critical System Lifecycle (SCSL), hazard analysis techniques, and model based testing. Section 2.1 explains the safety-critical lifecycle according to [IEC 61508] and explains its phases. The most common hazard analysis techniques such as FTA, PHA, FMEA, FMECA, HAZOP, ETA, CED, and FHA are explained in section 2.2. Section 2.3 elucidates model based testing techniques. The combined fault and behavioral models are discussed and illustrated with an example in section 2.4. The limitations of these approaches are explained in section 2.4.1.

2.1 Safety-Critical System Lifecycle (SCSL)

"Safety lifecycle models are considered to form an adequate framework to identify, allocate, structure, and control safety-related requirements" [IEC 61508]. This model is standardized in IEC 61508 to cover the complete safety lifecycle. It consists of sixteen phases as can be seen in Figure 2.1.

1. Concept.
2. Overall scope definition.

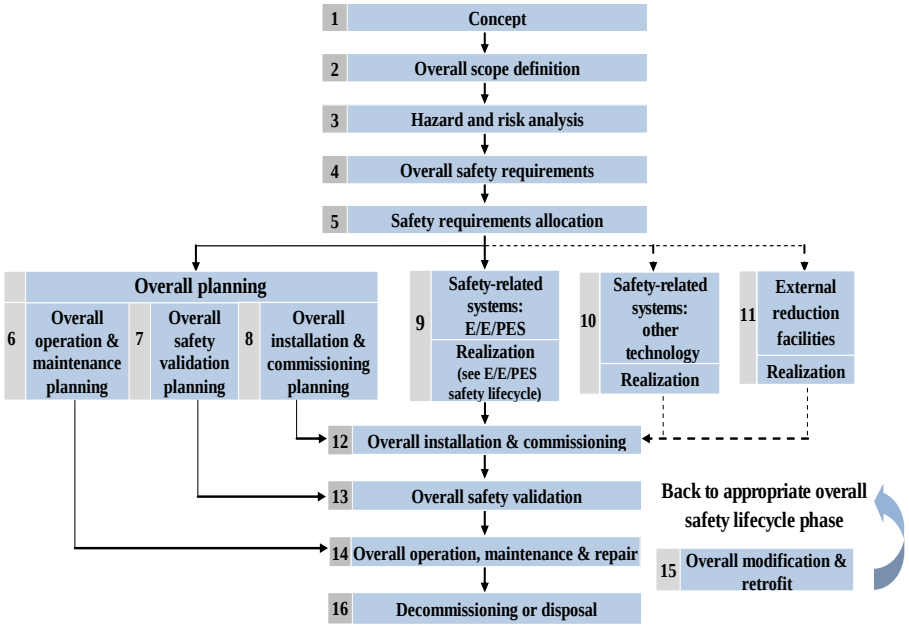

Figure 2.1: Overall Safety Lifecycle[IEC 61508]

3. Hazard and risk analysis.

4. Overall safety requirements.

5. Safety requirements allocation.

6. Overall planning: overall operations and maintenance planning.

7. Overall planning: overall safety validation planning.

8. Overall planning: overall installation and commissioning planning.

9. Safety-related systems (SRSs): Realization of Electrical, Electronic and Programmable Electronic systems (E/E/PE) SRSs.

10. Safety-related systems (SRSs): Realization of "other technology" SRSs.

11. Safety-related systems (SRSs): Realization of "external risk reduction facilities".

12. Overall installation and commissioning.

13. Overall safety validation.

14. Overall operation, maintenance and repair.

15. Overall modification and retrofit.

16. Decommissioning or disposal.

The safety lifecycle model is divided into three parts each of which contain some phases that address safety related issues. The first state contains the phases from phase 1 to phase 5, the second part contains the phase from phase 6 to phase 11 and the last part contains phases from 12 to 16. Every part is concerned with a step in the development of the safety related system lifecycle.

The first part of the lifecycle, phases 1, 2, 3, 4, and 5, concerns the risk analysis during which the potential hazardous situations are determined, their impact and consequences are established and the probability of occurrence estimated. Consequently, the need for additional risk reduction measures is determined and the safety requirements are specified and allocated to safety related systems.

The second part of the lifecycle, phases 6, 7, 8, 9, 10, and 11, concern the technical specification, development and implementation of the safety-related systems. As can be seen from the overall safety lifecycle model, phases 6, 7 and 8 are concerned with the overall planning. Their position emphasizes the importance of their overall status, even though in the standard they are defined as applying only to Electrical/Electronic/Programmable Electronic (E/E/PE) systems. Phases 9, 10 and 11 are concerned with the realization of the SRS, which may take the form of E/E/PE systems, other technology systems, or external facilities.

The third part, phases 12 to 16 concern the utilization of the SRS. During this part, requirements are defined concerning commissioning, operation, maintenance, periodic tests, eventual modifications and decommissioning of the SRS. These phases

demonstrate that the standard is not restricted to the development of systems, but that it covers the management of functional safety throughout a system's life. Many of the standard's requirements are indeed technical, but it is effective safety management rather than merely technical activities which in the long run must be relied on for the achievement of safe systems. Parts 2 and 3 of the standard address hardware and software development respectively for E/E/PE systems.

2.2 Hazard Analysis Techniques

The Hazard Analysis falls into the first part of the SCSL lifecycle. This part, at the early stages of the life-cycle, deals with the analysis of the hazards so that they can be considered in the coming stages. In this phase, preliminary Hazards and Operability (HAZOP) analysis [83] is performed along with Layers of Protection Analysis (LOPA) [130] and Criticality Analysis to know what the kinds of hazards are, how severe they would be, and how likely they may occur. This analysis allows for better understanding of the hazards to take the required actions in consideration in the coming stages of the lifecycle [43]. Table 2.1 contains the most common hazard analysis techniques, where they can be used in the development life-cycle, and whether they are quantitative or qualitative techniques. A complete list of these techniques can be found in [43].

- Fault Tree Analysis (FTA)

  Fault Tree Analysis (FTA) [143] is a safety analysis technique that is commonly used to analyze the safety of systems that are under development or are existing systems. It was originally designed by Bell Telephone Laboratories in 1962 for the US Air Force as a technique for the safety analysis of electromechanical devices and later used in analyzing safety-critical software [91]. It is

Table 2.1: Hazard Analysis Techniques

Technique	References	System Life-cycle Phase	Quantitative/ Qualitative
Fault Tree Analysis (FTA)	[91, 38, 77, 75, 76, 99, 32, 28, 131, 136, 124]	Preliminary Design & Detailed Design	Quantitative/ Qualitative
Preliminary Hazard Analysis (PHA)	[43, 58]	Conceptual & Preliminary Design	Qualitative
Failure Modes and Effects Analysis (FMEA)	[123, 56, 114, 66]	Preliminary Design & Detailed Design	Qualitative
Failure Mode, Effects and Criticality Analysis (FMECA)	[19, 146, 11, 23]	Preliminary Design & Detailed Design	Quantitative/ Qualitative
HAZard and OPerability (HAZOP)	[83, 39, 116, 30, 73]	Preliminary Design & Detailed Design	Quantitative
Event Tree Analysis (ETA)	[9, 79, 63]	Preliminary Design & Detailed Design	Quantitative/ Qualitative
Cause and Effect Diagrams (CED)	[81, 59, 42]	Preliminary Design, Detailed Design	Quantitative
Functional Hazard Analysis (FHA)	[149, 43]	Conceptual Design, Preliminary Design, and Detailed Design	Quantitative

a top-down safety analysis technique in which an undesired state of a system is analyzed using logical operations to combine a series of lower-level events.

A FT is composed of nodes, edges, and gates. Gates are logical connectors of events, while nodes represent events, and edges connect nodes to gates. When FT is used to model faults, every major hazard is represented by a separate fault tree.

Table 2.2: Fault Tree Gate Types [143, 40]

Symbol	Gate	Meaning
$\wedge$	AND	The gate occurs only when all its inputs occur.
$\overline{\wedge}$	PRIORTY AND	The gate occurs only when all its inputs occur in a specified order.
$\vee$	OR	The gate occurs when at least one of its inputs occurs.
$\hexagon$	INHIBIT	The gate occurs only when the input occurs and the enabling condition is true.
$\oplus$	XOR	The gate occurs only when the XOR of the inputs is true.
Single event gate	TIMING GATES	These gates occur only when an event occurs and the time-out is triggered.

Table 2.2 lists the gate types we consider here. A fault tree provides qualitative and quantitative measures of the likelihood of the occurrence of hazards [143]. Quantitative analysis is done by computing the probability of the occurrence of the root node from the probabilities of the lower level nodes, while Qualitative analysis shows the set of events that, if happen together, contribute to cause the hazard. Qualitative analysis is applicable when integrating faults into the system model because the analysis is performed on the

actual occurrence of the set of possible faults rather than on their probability of occurrence.

Kaiser *et al.* [77, 75] propose a compositional extension of the FTA technique. Each technical component in the system is represented by an extended Fault Tree that has, besides its basic events and gates, input and output ports. These components can be developed independently and can be integrated into a higher-level model by connecting these ports. Both qualitative and quantitative analysis can be applied on this FTA.

FTA [143] describes how the combination of behaviors of system components result in a hazard or a failure of a system. Although it is one of the most used techniques, it may not be suitable for software safety analysis because it is a static model that describes the overall cause of a hazard and cannot answer the questions why, when, and how the hazard occurs during the software execution. However, some work has been done to use FTA in testing safety-critical software. Miguel *et al.* [32] incorporate safety requirements in software architecture based on safety objectives, and evaluate these software architectures based on safety analysis methods such as FTA. The results of this incorporation are used to evaluate the architectures and to detect inconsistencies of software architectures and safety requirements.

Chen *et al.* [28] used FTA to evaluate the reliability of the railway power systems and investigates the impact of the maintenance in the reliability. A binary decision diagram (BDD) algorithm is used to evaluate the FT. Sun *et al.* [131] integrated FTA with Architecture Analysis and Design Language (AADL) to support the consistent reuse of FTA across the systems to reduce the effort of maintaining traceability between the safety analysis and the architectural models. Others such as Tracey *et al.* [136] integrate safety anal-

ysis with the automatic test-data generation to be used for software safety verification.

- Preliminary Hazard Analysis (PHA)

  It is a comprehensive, structured, and logical technique for identifying and evaluating risk in complex technological systems that produces detailed identification and assessment of accident scenarios [58]. PHA is an activity that takes place while developing the system design to identify software-related hazards. Therefore, these hazards or its consequences can then be removed [43].

  PHA helps ensure that the system is safe and makes the system modifications less expensive and easier to implement in the earlier stages of design. Moreover, it decreases design time by reducing the number of surprises and unexpected outcomes. On the other hand, in PHA hazards must be foreseen by the analysts so they can deal with them. However, foreseeing hazards may not be an easy task since the effects of interactions between hazards are not easily recognized.

- Failure Modes and Effects Analysis (FMEA)

  FMEA is a fault analysis technique that aims to identify hazards in requirements that have a potential failure [123, 56]. It is a bottom-up technique that can be applied during the analysis and design phases. It is used to identify critical functions based on the applicable specification. The severity and the likelihood of a mishap will be used to define the criticality level of the function and thus it will be considered more deeply in a later criticality analysis [114]. Processing FMEA manually can be an error-pron, costly, and hard to repeat

process. To avoid these drawbacks, Hecht *et al.* in [66] automate the major steps in generating a software FMEA.

The FMEA technique has the capability to identify and eliminate potential failure modes early and thus reduces the cost associated with late changes. It also reduces the possibility of the occurrence of the same failure in the future. On the other hand, this technique may only identify major failure modes in a system and is limited by the analyst's experience of previous failures.

- Failure Mode, Effects and Criticality Analysis (FMECA)

  FMECA extends FMEA by including a criticality analysis. It charts the probability of failure modes against the severity of their consequences. The FMECA process should be initiated as a part of the early design process and should be updated to reflect design changes as FMECA is a major consideration at Design Review [146]. Due to the nature of the design process, the FMECA must also be iterative. The purpose of a FMECA is to provide a systematic, critical examination of potential failure modes and their causes, assess the safety of systems or components, analyze the effect of each failure mode, and identify corrective action. It has been used in analyzing safety in safety-critical systems with software systems in the aerospace [23, 11] and automobile domains[19].

  FMECA is a systematic comprehensive technique that establishes relationships between failure causes and effects. It has the ability to point out individual failure modes for corrective action in design. However, due to its comprehensiveness, a large number of trivial cases will be considered which requires extensive work. FMECA is unable to deal with multiple-failure scenarios or unplanned cross-system effects such as sneak circuits (conditions

which are present but not always active, and they do not depend on component failure [69]).

- HAZard and OPerability (HAZOP)

  HAZOP is an analysis technique that assumes the deviations from the design or operating intentions cause accidents. Therefore, it puts in consideration all possible ways that the hazards or operating problems may arise if the system is operated under a different mode than the intended operating conditions. The concept of a HAZOP study first appeared with the aim of identifying possible hazards present in facilities that manage highly hazardous materials [83]. The purpose was to eliminate any cause of major accidents, such as toxic releases, explosions, and fires. Because of its success in identifying hazards and operational problems, HAZOP's application extended to other types of facilities. Therefore, it was adopted for computer-based systems such as medical diagnostic systems [30, 39] and in railway systems [116, 73].

- Event Tree Analysis (ETA)

  ETA is a technique used to explore responses to an initiating event and enables assessment of the probabilities of outcomes [9]. It is a bottom-up approach used to define potential accident sequences associated with a particular event or set of events. It was first applied in risk assessments for the nuclear industry and then later was adapted by others such as chemical processing, offshore oil and gas production, transportation, and safety critical software. The Event-Tree can be used as a quantitative and qualitative analysis technique. Quantification of the event-tree diagram allows the frequency of each of the outcomes to be predicted.

ETA is structural, rigorous and, a large portion of it can be computerized. It models complex systems relationship in an understandable manner and can be performed on many levels of design detail. Moreover, it combines hardware, software, environment, and human interaction and it permits probability assessment. On the other hand, ETA can not distinguish between partial successes and failures. It also requires an analyst with some training and practical experience and can only have one initiating event. In addition, when modeling an event, subtle system dependencies can be overlooked.

- Cause and Effect Diagrams (CED)

  CED [81], also known as Ishikawa Analysis, is a quantitative analysis that graphically represents the relationships between a problem and its possible causes. The advantage of this analysis technique is that it is very simple, visual and easy to understand by the analysts. The drawbacks of CEDs are: (1) there is a lack of distinction between necessary conditions and sufficient conditions and (2) not all the logical possibilities of the occurrence of the causes are taken into account [59]. The cause in this analysis technique is broken down into other causes and these can also be broken down into other causes. Therefore, it is difficult to understand what the word 'cause' exactly means. Does it mean a necessary condition, a sufficient condition, or a necessary and sufficient condition [42].

- Functional Hazard Analysis (FHA)

  FHA is defined as one of the preliminary activities in the safety assessment process [149]. It is a quantitative analysis technique for identifying all the hazards that can affect the outcome of the principal functional activities that need to be carried out to accomplish a given task [149]. The purpose of

FHA is to identify system hazards by the analysis of functions. Functions are the means by which a system operates to accomplish its tasks. System hazards are identified by evaluating the safety effects of a function failing to operate, operating incorrectly, or operating at the wrong time. They may consist of a loss of critical function, inadvertent activation of the function, outside influences on the performance of the function, or some combination of them. When a failure of a function is determined, the cause factor should be investigated in more detail.

The advantages of this technique are: (1) this technique works best for functions that are entirely independent; (2) it helps to better understand the effect of a failure. The drawbacks of FHA are: (1) It is hard to identify functions at the right level of abstraction from the available requirements, (2) determining the effect of function failure of lower level function can be difficult, and (3) it is hard to apply FHA for dependent functions.

There are many safety analysis techniques that have been in use for many years. These techniques are used during the development stage of the system lifecycle. Some of these techniques are used at the system level to analyze the system hazards, while others are used at the subsystem levels to define the hazards related to components and their interaction. The results of hazard identification help the designers and the developers eliminate and mitigate these hazards.

2.3 Model Based Testing (MBT)

Model-based Testing uses behavioral models of the software produced from the functional requirements to carry out the software testing activity. FSM, UML, EFSM, and CEFSM are common modeling techniques used in modeling software

systems. They have been used for testing activities such as test case generation as well. Many coverage criteria such as edge, node, edge-pair, prime path, and W-method were imposed on one or more of these models and each of these criteria satisfies one or more test requirements. Table 2.3 shows the models and the types of systems for which they were used. The rows contain the modeling techniques such as UML activity diagram, UML statechart diagram, and FSM, while the columns consist of some types of the systems such as non-embedded, embedded, automotive, and aerospace.

Table 2.3: Model Based Testing

Model		Not embedded	Embedded	Automotive	Aerospace
UML	Activity	[27]	[94]		
	Statechart	[109, 110, 21]	[103, 96, 147]		
	collaboration	[2]			
	Sequence	[25]	[128, 92, 104]		
	class	[122]			
	use case	[133, 102]			
	combination	[64, 13, 12, 14, 97]	[150]	[106, 45, 37, 65, 84]	[153, 35, 115]
FSM		[29, 7, 72, 70, 3, 89]	[107, 47, 100]	[113]	
EFSM		[118, 34, 44, 61, 62, 139, 141]	[134]	[152]	[78]

2.3.1 Unified Modeling Language (UML)

UML [125] is the de facto standard language for specifying, modeling, analyzing, and documenting software [117]. It is also used in modeling hardware, in business contexts as well as in modeling systems. Its graphical notations makes it easier to express and understand the design of software systems. It has also been used in testing system implementations against their design artifact [22, 21, 55, 87], or used to test the design of the system itself [57, 120, 36, 121]. In this section, we will discuss the work used UML in testing.

The Statechart diagram is one of the UML diagrams that is expressive, representative, and easy to use by the modeler. It consists of *States* which allow for hierarchy to support the scalability of the representation. Thus states can contain other states which can be represented as AND, or OR states. This means that a state can have an orthogonal decomposition, OR decomposition, or have no child states. In addition to States, a statechart contains *Events* that represent the occurrence of happenings that may trigger a *Transition* which shows what the next state will be. *Parameters* of an event are global variables that can be used to convey quantitative information regarding that occurrence and can be used by *Guard conditions* to enable actions or transitions only when they evaluate to true. Figure 2.2 shows a small example of a statechart that contains two states, (*Start* and *PartialDail*), an event (*digit*), a parameter (n), a transition is the arrow between the states, and a guard (*[Number.isValid()]*).

UML statechart diagram has been used for testing the system as a whole or as components or functions for non-embedded as well as embedded software systems. Offutt *et al.* in [109] present a technique to generate test cases from UML statecharts. This technique adapts the state-based specification test data generation

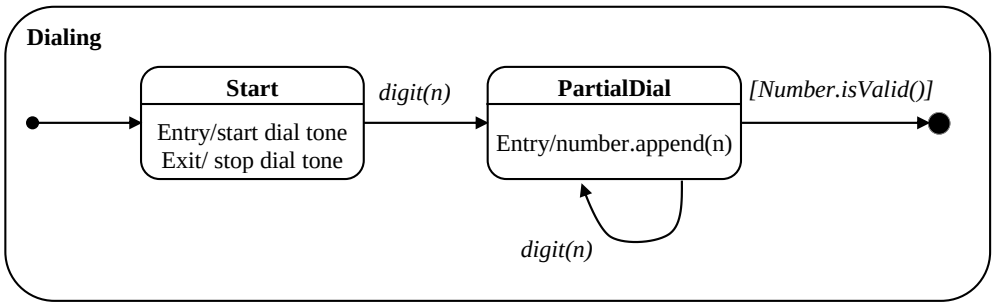

Figure 2.2: Statechart Example

criteria presented in [108]. Offutt *et al.* in [110] present general criteria for generating test inputs from the state-based specifications proposed in [109] which are transition coverage criterion, full predicate coverage criterion, transition-pair coverage criterion and complete sequence criterion. It is possible to apply all these criteria or to choose any one of them based on a cost/benefit tradeoff.

Many statechart based testing strategies require flattening the statechart to specify a set of paths to be executed. These techniques can be automated. From the flattened statechart, we can take each path and produce a test case. Briand *et al.* [21] propose a methodology to automate the procedure of generating test case from a statechart using coverage criteria such as all transitions, all transition pairs, full predicate, and all round-trip paths. This methodology assumes that a test case to be in the form of a feasible sequence of transitions. The procedure is to take each test path separately and derive test data. This requires identifying the system state involved for each event/transition that is part of the path to be tested and the input parameter values for all events and actions associated with the transitions. They introduce a number of algorithms to generate test constraints automatically.

Leftcaru *et al.* [89] use genetic algorithms (GA) to generate test data for chosen paths in the state machine, so that the input parameters provided to the methods trigger the specified transitions. The GA searches for the input parameters which

satisfy the specified requirements. After this technique obtains some paths according to some coverage criteria, it then finds, for each path, the input parameter values that trigger the methods in that path. Murthy *et al.* [103] introduce Test-Ready UML statechart models to be used for testing. This model is obtained by annotating the statechart model with events, guards, conditions, tasks and test statements along the transitions. They also made the test generation automatic from a UML statechart by identifying the required annotations for the UML statecharts. The test path generation algorithm they used is based on depth-first traversal of the model.

Lochau *et al.* [96] generate test cases that aim at feature interaction analysis by using a UML statechart diagram. In order to automatically generate test cases, they defined the components' dynamic behavior via UML statecharts, specified the interactions amongst these components, and annotated the test requirements. Test cases are then derived from these annotated statecharts. Weibleder *et al.* [147] compare several approaches for generating test cases from a UML statechart based on a set of quality goals or metrics that are used to determine when to stop testing, instead of testing until the available resources are exhausted.

The Activity diagram is another UML diagram. It describes dynamic aspects of the system. It is a flow chart that represents the flow from one activity to another. It consists of four basic elements: (1) rounded rectangles represent *Actions* that are part of an activity diagram, (2) a black circle represents the start *initial state* of the workflow, (3) diamonds represent *decisions*, (4) bars represent the start *split* or end *join* of concurrent activities, and an encircled black circle represents the end *final state*. Figure 2.3 illustrates a simple example of an activity diagram. The UML activity diagram has also been used as a testing model. The work of Linzhang *et al.* [94] generates test cases directly from a UML activity diagram using a gray-box

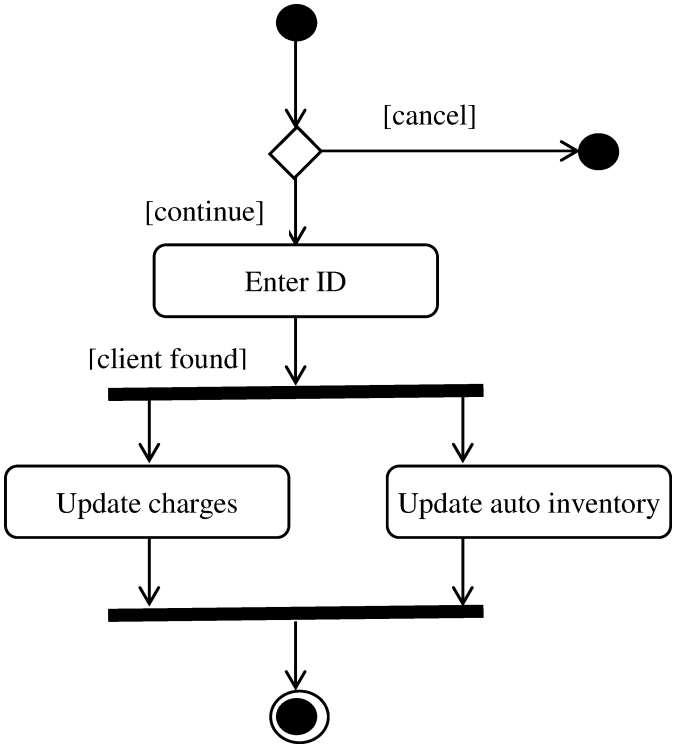

Figure 2.3: Activity Diagram Example

method. A gray-box method is a combination of white-box and Black-box methods. A Gray-box method generates test cases by parsing the activity diagram to derive the set of test scenarios that satisfies the path coverage criteria by applying depth first search on the activity diagram. Input and output parameters are then extracted from each test scenario. Test cases are then obtained from the input and output sequences, guards and constrains.

Mingsong *et al.* [27] also used UML activity diagrams as design specifications to generate test cases. The approach randomly generates numerous test cases for a program under test. Then, they execute the program with the generated test cases to obtain the corresponding outputs. After that, they compare these outputs with the given activity diagram according to the specific coverage criteria to obtain a reduced test case set that meets the test adequacy criteria.

Collaboration diagrams describe a collection of objects that interact to implement some behavior within the context of the system. They illustrate the rules of the objects in a system and how they communicate to perform a specific task according to a use case. The basic elements of collaboration are *ClassifierRoles* which describe how objects behave, *AssociationRoles* that describe how an association will behave in a particular situation, and *Interactions* which represent operations/methods that the receiving object's class implements. A message defines a particular communication between instances that is specified in an interaction. Messages in a collaboration diagram are numbered in the order of the execution.

Collaboration diagrams were used in testing by Abdurazik *et al.* [2] with test criteria. Test cases are generated from the collaboration diagrams according to these criteria. Each collaboration diagram represents a sequence of messages that corresponds to a use case. These criteria allow formal integration tests to be based on high level design notations.

Sequence diagrams model the cooperation of objects relying on a time sequence. They are known as event diagrams, event scenarios, or timing diagrams [1, 46]. They show how objects interact with one another in a particular scenario of a use case. Sequence diagrams capture the invocation and the occurrence order of methods from each object. A sequence diagram consists of: (1) Object which is the a primary element involved in the diagram and represented by a rectangle, (2) Message is the interactions between different objects in a sequence diagram and. A message is denoted by a directed arrow and the notation differs depending on the message type. A complete arrow is used for check and assignment statements, identifying the nature of the operation as a comment, while a normal arrow is used instead for the activation of an operation.

Cartaxo et al. [25] generate test cases from UML sequence diagrams based on the derivation of Labeled Transition System (LTS). The LTS [80] provides a global monolithic description of the set of all possible behaviors of the system. A path on the LTS can be taken as a test sequence. The Depth First Search technique (DFS) is used to obtain a path by traversing an LTS starting from the initial state. They use state and transition coverage criteria to generate test cases. The transformation from UML sequence diagram to LTS is done by Unified Modeling Language All pUrposes Transformer (UMLAUT) tool and the test case generation by Test Generation with Verification technology (TGV) tool. The procedure is targeted to feature testing of mobile phone applications whose requirements are specified by sequence diagrams, including loops and alternative flows.

Sarma *et al.* [128] introduce a method of generating test cases from a UML sequence diagram by transforming a UML sequence diagram into a graph called the Sequence Diagram Graph (SDG). Each node in the SDG stores information necessary for test case generation. This information is collected from the use case template, class diagrams, and data dictionary represented in the form of object constrained language (OCL), that ware associated with the use case for which the sequence diagram is considered. The SDG is traversed and test cases are generated using all message path sequence coverage criteria. They generate test cases that satisfy the criteria by first enumerate all possible paths from the initial node to the final node in the SDG. Each path then would be visited to generate test cases.

Bao-Lin *et al.* [92] introduce a test cases generation approach which relies on UML sequence diagrams and Object Constraint Language (OCL). They represent sequence diagrams as tree by constructing a scenario tree (ST), and obtain the scenario path from ST. Then, they use the message path coverage and constraint

attribute coverage to generate test cases. They iteratively select all messages from SD and use OCL to describe the pre and post conditions.

Nayak *et al.* [104] introduce an approach of synthesizing test data from the information embedded in model elements like class diagrams, sequence diagrams and OCL constraints. They develop a sequence diagram with attribute and constraint information derived from the class diagram and OCL constraints and map it onto a Structured Composite Graph (SCG). Test paths are then generated from SCG using all message criteria. They generate test data for each test path by following a constraint solving system. The proposed approach assume that initially all test paths are feasible unless it cannot be exercised by any set of input data and then the path becomes infeasible. Many works have used more than one UML diagram for testing as well. Bertolino *et al.* in [14] develop a framework for test derivation and execution in a component-based development environment by integrating some existing tools and methodologies. The UML components methodology is used to define the diagrams necessary to apply the Cow_Suite tool (Cow_Suite is a UML-based test environment for test-suite planning and derivation). Then the tests are composed and executed within the Component Deployment Testing CDT framework.

Many UML diagrams such as interaction diagrams, statechart diagrams, and component diagrams, have been used to characterize the behavior of components in various aspects, so that they can be used to test component-based systems. Bertolino *et al.* [13] combines sequence and state diagrams in order to produce a more informative testing model. The resulting model is used to identify more accurate test cases. In case that the sequence model is conformant with the state model, the state model will be used as a reference model in order to generate further test cases. This work is meant to improve the work proposed in [14]. Their goal is produce from

the incomplete diagrams a more complete model to extract test cases from without requiring extra modeling effort.

Zoughbi *et al.* [153] propose an UML profile to improve the communication between safety engineers and software engineers. A UML profile will allow software engineers to model safety related concepts and properties in UML. Safety-related concepts are extracted from RTCA DO-178B (the airworthiness standard is the software standard for commercial and military aerospace programs). Then, the UML profile is presented to enable modeling these safety-related concepts. Supakkul *et al.*[132], Mayer *et al.* [102], and Donini *et al.* [37] propose different approaches to generate test cases from UML sequence and activity diagrams by first transforming these diagrams into a graph, generating test scenarios from the constructed graph and then extracting the necessary information for test case generation.

Swain *et al.* [133] integrate UML sequence and activity diagrams to generate test cases. This is done by transforming these UML diagrams into a graph. An algorithm to generate test scenarios from the constructed graph is then applied. Next, the necessary information for test case generation, such as method-activity sequence, associated objects, and constraints and conditions are extracted from the test scenario. This approach is meant to reduce the number of test cases while achieving adequate test coverage. Wu *et al.* in [150] defines some UML-based test adequacy criteria that can be used to test component based software.

Hartmann *et al.* in [64] models components with their interactions and derived test cases are from these component models and then execute them to verify their conformant behavior. Prasanna *et al.* in [122] derive test cases by analyzing the dynamic behavior of the object diagrams (a detailed state of the system at a point of time) taken from the UML model of the system. This diagram is mapped to a tree. Genetic algorithm's crossover technique is then applied to this tree to generate

a new generation of trees. Each tree is then converted into a binary tree and a depth first search technique is applied on these binary trees to produce test cases.

UML techniques have also been used in testing automotive and aerospace systems. Flamini *et al.* in [45] present a methodology to automatically perform an 'abstract testing' of a large control systems. The abstract testing can be defined as a configuration-independent and auto-instantiating approach for large computer-based control systems. It is specified from system functional requirements and covers many system configurations. It can be instantiated to cover any number of control entities (sensors, actuators and logic processes). The configuration of the system is used as an input to the transformation algorithm from abstract to specific tests that are suitable for this configuration. The algorithm executes the test cases one by until a failure is found or the test suite is set is empty. This approach saves a considerable time effort required for this process when it is performed by hand. In the same field Nicola *et al.* in [106] describe the Ansaldo Segnalamento Ferroviario (ASF) functional testing methodology, based on a grey-box approach to generate and reduce an extensive set of influence variables and test-cases. An influence variables is a variable that effects the behavior of the system under test.

2.3.2 Finite State Machine (FSM)

FSM has a long and rich history as a modeling and testing language [4]. It has been used for testing activities such as test case generation [4]. Many coverage criteria such as edge, node, edge-pair, prime path, and W-method were imposed on the FSM model and each of these criteria satisfies one or more test requirements. A FSM is defined as: [88]

M is a quintuple $M = (I, O, S, \sigma, \lambda)$ where

- I is a finite and nonempty set of input symbols,
- O is a finite and nonempty set of output symbols,
- S is a set of states,
- δ: S$\times$ $I \rightarrow S$ is the state transition function, and
- λ: S$\times$ $I \rightarrow O$ is the output function.

When the machine is in a current state $s \in S$ and receives an input $i \in I$ it moves to next state specified by $\delta(s,\ i)$ and produces an output given by $\lambda(s,\ o$ where $o \in O$.

Chow [29] proposed a testing strategy known as "automata theory" or "W-method". It verifies the correctness of control structures that can be modeled by FSM. This strategy showed that it can find transition errors, state errors, and operation errors. Therefore, it can be applied to software testing and the test results derived from the design are evaluated against the specification. Fujiwara *et al.* [48] presented a new method called partial W-method (Wp). This method is a variation of that proposed in [29] that provides shorter test sequences than the W-method.

Luo *et al.* [98] studied the issue of test selection for open distributed processing with several distributed interfaces that was modeled in FSM. They also developed a test generation method to generate test sequences based on the idea of synchronizable test sequences. Tsai *et al.* [138] presented an approach to automatically generate test cases for an object oriented class. These test cases are generated based on the test case tree which is built based on an implementation state machine which in turn is built up from the design state machine and the implemented class.

Friedman *et al.* [47] generate tests based on FSM models of a specification. They describe a set of coverage criteria and testing constraints that compromise between state and transition coverage criteria, and input domain coverage criteria.

The transition and state coverage often lead to large suite of test cases while the input domain requires big engineering efforts.

2.3.3 Extended Finite State Machine (EFSM)

EFSM is an extension of the original FSM. The expressiveness power of EFSM makes it capable of modeling system specification that include variables and operations based on variable values. In an FSM, the transition is associated with a set of inputs and a set of output functions, whereas in an EFSM model, the transition will be fired if the predicate conditions are all satisfied, moving the machine from the current state to the next state and performing the specified data operations.

An EFSM is 5-tuple = *(S, I, O, T, V), such that:*

- S is a finite set of states,
- I is a set of inputs symbols,
- O is a set of output symbols,
- T is a set of transitions,
- V is a set of variables, and

State changes: The transition t in the set T is a 6-tuple:

$t = T(s_t, \acute{s}_t, i_t, o_t, P_t, A_t)$ where,

- s_t is the current state,
- $\acute{s}_t$ is the next state,
- i_t is the input,
- o_t is the output,
- $P_t(\vec{v})$ is predicates on the current variable values, and
- $A_t(\vec{v})$ is the action on variable values.

EFSM has also been used for software testing. Tahat *et al.* in [134] automatically generates a system model from the requirement information. This model is then used to automatically generate test cases related to individual requirements. This approach is extended to generate regression tests that are related to the requirement changes. Derderian *et al.* in [34] use a genetic algorithm to create an input sequence that triggers a given path within an EFSM. Fantinato *et al.* in [44] extend the FSMs to provide data flow modeling mechanisms to be used as a basis to define a set of functional testing criteria.

Kalaji *et al.* [78] developed an approach to optimize the testing from EFSM. The aim of this approach is to overcome the path feasibility and path test data generation problems. A path is said to be infeasible if there is no input data that can trigger such path. This is due to the fact that the transition in a EFSM model includes predicates and operations that there does not exist data can trigger such path. However, finding such a set of input data for the feasible path is difficult task since the input domain is usually large and the required values is a small subset of this domain. A fitness metric is used to estimate the likelihood of the feasibility of a given path. EFSM transitions and their input parameters can be considered as functions and their input parameters. The fitness function is used to guide the search for a suitable set of inputs.

Guglielmo et al. in [61] use the extended finite state machine (EFSM) model to generate test sequences. The same author *et al.* in [62] propose a functional deterministic automatic test pattern generation (ATPG) approach that uses EFSMs for functional verification.

2.3.4 Communicating Extended Finite State Machine (CEFSM)

CEFSM is an extended type of the traditional EFSM that provides data flow modeling and communications channels. CEFSM F is a tuple $\langle E_F, C_F \rangle$, where E_F is an EFSM and C_F is a set of input/output communication channels used in this CEFSM. CEFSM has been used in modeling and testing distributed systems and network protocols. The strength of CEFSM is that it can model orthogonal states of a system in a flat manner and does not need to compose the whole system in one state as in statecharts. which would makes them more complicated and harder to analyze and/or test. Communicating EFSMs can be defined as a finite set of consistent and completely specified EFSMs along with two disjoint sets of input and output messages[88]:

CEFSM = (S, s_0, E, P, T, M, V, A, C), such that:

- S is a finite set of states,
- s_0 is the initial state,
- E is a set of events,
- P is a set of boolean predicates,
- T is a set of transition functions such that T: S×P×E→S× A×M,
- M is a set of communicating messages,
- V is a set of variables,
- A is the set of actions, and
- C is the set of input/output communication channels used in this CEFSM.

State changes (action language): The function T returns a next state, a set of output signals, and action list for each combination of a current state, an input signal, and a predicate. It is defined as:
$T(s_i,\ p_i,\ get(m_i))/(s_j,\ A,\ send(m_{j_1},\ldots,\ m_{j_k}))$ where,

- s_i is the current state,
- s_j is the next state,
- p_i is the predicate that must be true in order to execute the transition,
- e_i is the event that when combined with a predicate trigger the transition function,
- $m_{i_1},\ldots,\ m_{i_k}$ are the messages, and

The communicating message m_i is defined as:
(mId, e_j*, mDestination)* where,

- *mId* is the message identifier, and
- *mDestination* is the CEFSM the message is sent to.

An event e_i is defined as: *(eId, eOccurrence, eStatus)* where,

- *eId* is the event identifier that uniquely identifies it, and
- *eOccurrence* is set to false as long as the event has not occurred for the first time and to true otherwise, and
- *eStatus* is set to true when the event occurs and to false when it no longer applies. Note that *eStatus* allows reoccurring events to happen multiple times (loops in the model).

CEFSMs communicate by exchanging messages through communication channels C that connect the outputs of one CEFSM to the input to other CEFSMs. Let

C denote the set $\{\langle name, SYNC|ASYNC\rangle|$ for all the channels in the system$\}$ where *name* is the name of the communication channel and *SYNC* and *ASYNC* indicate that the channel is synchronous or asynchronous. A same communication channel can be used differently according to different transitions. A channel c $\in$ C can be represented as $\langle name, t, get()/send()\rangle$ where,

- *name* is the name of the channel,
- $t \in$ T refers to the transition linked to this use of the channel, and
- *get()/send()* indicates whether this channel is an input or an output channel.

The action a_i may include an assignment and mathematical operations on the variables. The predicate is a condition that must be met prior to the execution of the function. For example, *T(S_0, e_0, total = 4)/(S_1, {m_0, m_1}, (total = 0; increment(i)))* describes that if a CEFSM is in a state S_0 receives an event e_0 and the value of variable *total*, which is the predicate, is four at that time, it will move to the next state S_1 and outputs m_0 and m_1 after setting the *total* to zero and performing *increment(i)*. For full formal semantics see [20].

2.3.4.1 Test Case Generation from CEFSM model

CEFSM-based test generation methodology proposed by Li *et al.* in [93] uses FSMs to model behavior and events. The extension of events with variables is used to model data while the events' interaction channels are used to model communication. The tests are generated based on a combination of behavior, data, and communication specifications. This method addresses branching coverage for data-related decision coverage and behavioral transition coverage. It applies priority and dominator analysis to generate efficient test cases to increase the branching coverage as much as possible with as few tests as possible. During the generation of test

cases, the priority of each branch is calculated by sorting them in decreasing order of additional branching coverage, while a dominator means that a node A is said to be a dominator of a node B if covering of B implies the covering of A. A branch can be a data-related decision or an event alternative, i.e., each branch is defined as a unique transition from a state to another state.

Hessel *et al.* [68] present an algorithm for generating test suites that cover all feasible coverage criteria. The algorithm is inspired by reachability analysis. The algorithm, at any given point, uses the information about the total coverage of the currently generated state space to avoid unnecessary state space exploration and to improve the performance of the algorithm. Derderian *et al.* [33] outlined the problem of observing local transitions of individual CEFSMs within a global transition (the interconnected transitions within the set of CEFSMs) using genetic algorithms without the use of a product machine. A product machine is a FSM that results from converting a set of CEFSMs.

The easiest approach to testing CEFSMs is to compose them as one machine at once, using reachability analysis, and then generate test cases for the product machine. However, this approach is impractical due to the state explosion problem and the presence of variables and conditional statements. Bourhfir *et al.* [16] propose generating test cases for systems modeled by CEFSM. Test cases can be generated for the global system by performing a complete reachability analysis, i.e., taking all transitions of all CEFSMs into consideration to generate test cases for the whole produced graph. CEFTG can also be used to generate test cases for each CEFSM individually. The algorithm terminates when the coverage achieved by the generated test cases is satisfactory or after the generation of the test cases for the partial products of all CEFSMs.

Kovács *et al.* in [86] methods and mutation operators are designed to enable the automation of test selection in a CEFSM model. These mutation operators do not simulate the typical errors of the specification or the implementation, rather they create erroneous specifications that provide the basis for test case selection.

2.4 Combined Behavioral and Fault Models

Testing safety-critical software differs from testing non-safety-critical software in many ways. Before testing safety-critical software systems, we need to conduct a safety analysis for the system to find possible safety breaches and what may cause them. The result of such an analysis is then used during testing. Another difference is that in safety-critical software we are testing for safety breaches, which is undesired behavior, in addition to the desired behavior. However, recent model-based testing techniques do not adequately consider the information derived from the safety analysis like Failure Mode and Effect Analysis (FMEA) and Fault Tree Analysis (FTA) [85]. Hence, people realized that there is a considerable gap between the safety analysis models and the behavioral models that needed to be bridged. Therefore, different approaches to integrate the fault analysis and system models were proposed and used in safety analysis and testing.

Al-Ariss *et al.* in [41, 40] integrate fault-tree-based safety analysis into a functional model. They use systematic transformation steps from Fault Tree to a statechart model. The integration results in an integrated functional and safety specification (IFSS) model that preserves the semantics of both the fault tree and the statechart. It also shows how the system behaves when a failure condition occurs.

Thus, it is used as a model that ensures safety through requirement validation. Example 2.4.1 illustrates a model of a microwave oven which will be applied for the next integration techniques.

Example 2.4.1. ***Microwave System:*** Figure 2.4 shows the behavioral model of the microwave system. The statechart model consists of the Timer, the Door, the Switch, and the Control as orthogonal regions. The door can be in one of the states open, opening, closing, or closed. The Switch is either switched on or off and the timer is either timing or idle, and the microwave can be in one the states, off, idle, or microwaving.

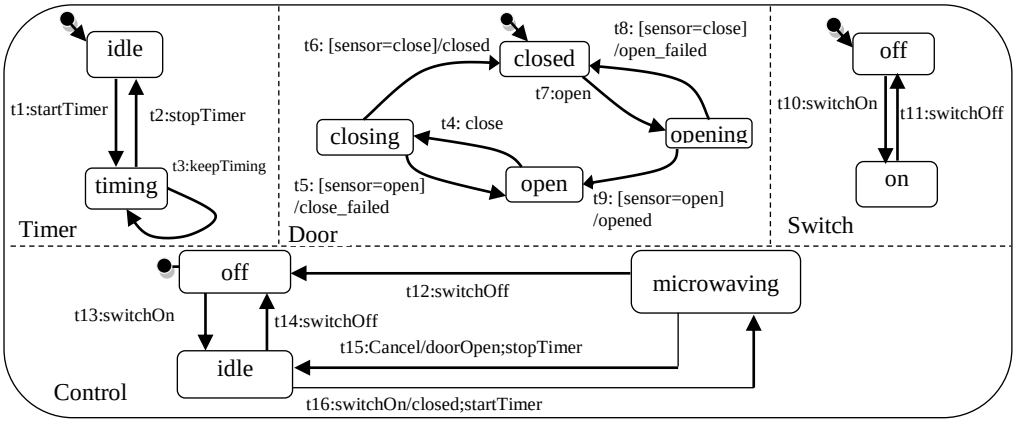

Figure 2.4: Statechart for Microwave System

Figure 2.5 illustrates the fault model for the exposure of microwave. This FTA shows the events that contribute in the top event *exposure of microwave*. For simplicity, we assume that the microwave oven is already switched on and the *Control* region is in the idle state. We used this small FTA that contains only three primary events which are *Timing*, *Door sensor failure*, and *Door open*. All these events are connected with the logical AND gate, which means all these events must occur in order for the top event to occur.

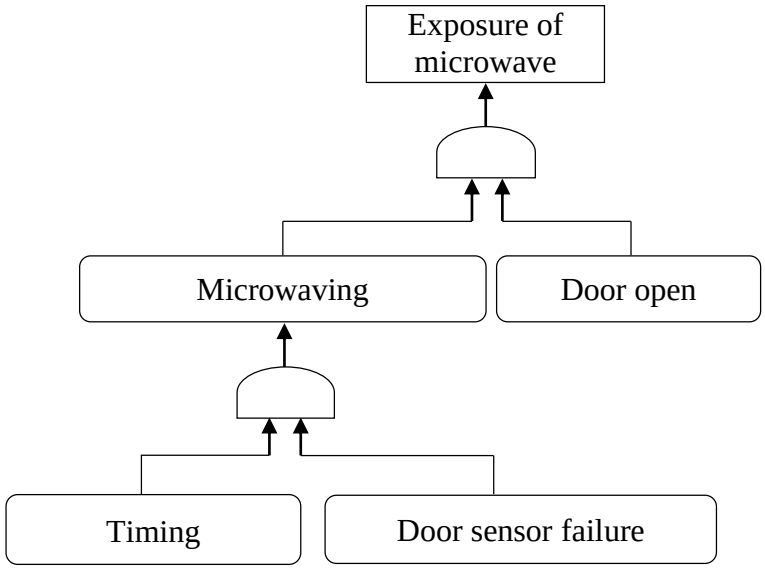

Figure 2.5: FTA for Microwave Exposure

The top event of the FTA is the event *exposure of microwaving*. This event is considered a mishap and composed of a combination of events. It occurs when the microwave oven door is open and the microwave is operational. This will expose the user to microwaves. The microwave is not supposed to operate when the door is open, however, this may happen if the door sensor gives a wrong signal or information to the controller.

We apply this technique [40] on Example 2.4.1. The transformation starts with deducing the safety requirements from the FTA and that will give the following formula:

$Exposure\ of\ microwave = (\wedge, (\wedge, (Timing, Door\ Sensor\ Failure), DoorOpen))$

First, we need to check whether the leaf nodes are simple or composed. In this example, all the leaf nodes are simple, therefore, the formula will not change. The next step is to deduce the library of semantics from 2.5 and 2.4, which is shown in table 2.4

Table 2.4: Semantic Table

Simple definition	Statechart component	Equivalent transition
Timing	Timer	(Timer.idle, t1, Timer.timing) (Timer.timing, t2, Timer.idle)
Door sensor Failure	Door	(Door.opening, t8, Door.closed) (Door.opening, t9, Door.opened) (Door.closing, t5, Door.opened) (Door.closing, t6, Door.closed)
Door open	Door	(Door.closed, t7-t9, Door.opened) (Door.opened, t4-t6, Door.closed)

After the semantic table is constructed, the transformation of FTA gates, gate by gate, starts from the top event to the leaves. At the beginning, we have two inputs from the top AND gate and they are *A1* and *A2* as shown in figure 2.6. The formula for *A1* and *A2* is

Exposure of microwave = (∧, (microwaving, Door Open).

Exposure of microwave = (∧, (A1, A2).

The equivalent statechart is shown in figure 2.7.

The formula is *Microwaving = (∧, (Switched on, Door sensor failure))* which is the same as *Microwaving = (∧, (A1.1, A1.2))*. See figure 2.8.

After the transformation of all the FTA gates, these gates will be integrated into the statechart of the behavioral model. The control region will be modified so that it includes the top event of the FTA. The IFSS model is shown in Figure 2.9. This model integrating approach has integrated a statechart and a fault tree according to some transformation and integration steps. The IFSS model can be used for safety analysis only. Our goal in this dissertation is to integrated behavioral model with

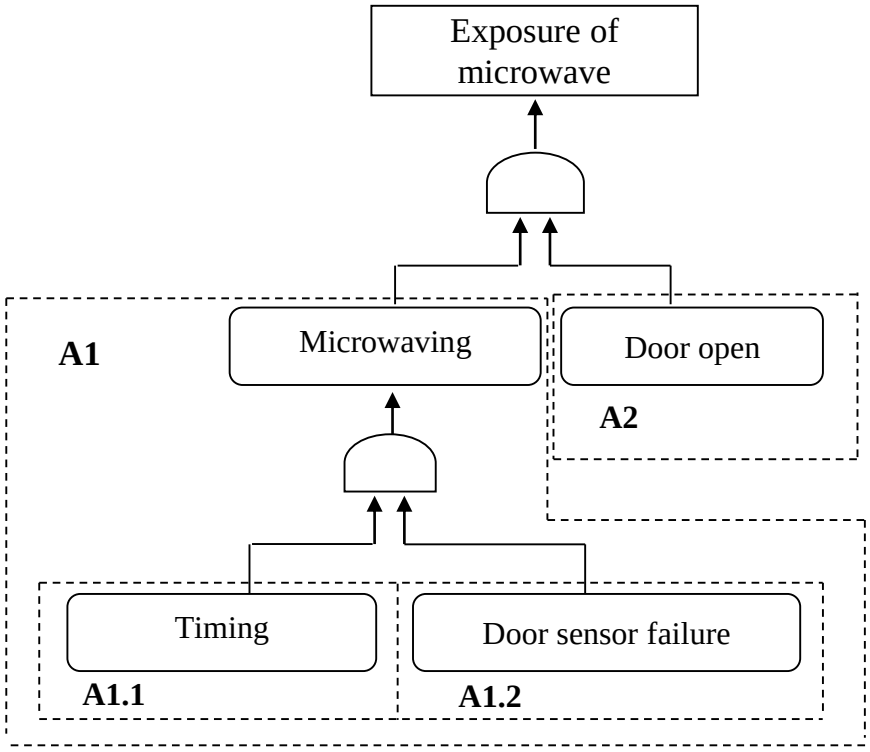

Figure 2.6: Applying Transformation Rules on Exposure of Microwave

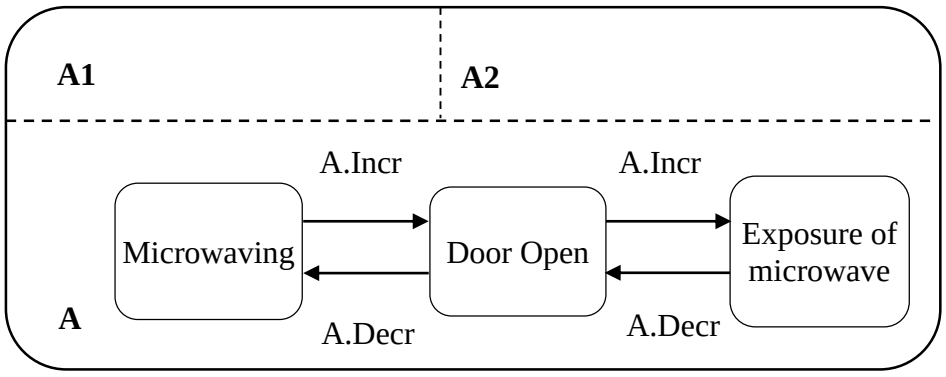

Figure 2.7: The Statechart Gate for Exposure of Microwave Event

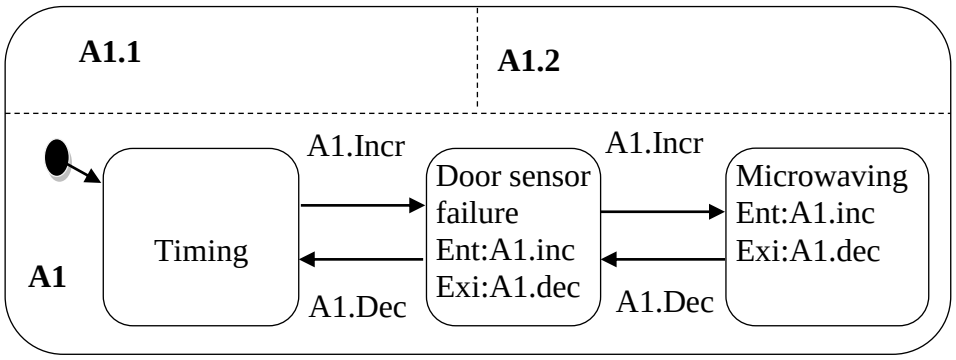

Figure 2.8: Statechart Gate for Microwaving Event

failure models to be used for safety testing. Kim *et al.* in [82] develop an algorithm to transform hazards of a Fault Tree (FT) into a UML statechart diagram in order to perform safety analysis. In this approach, the authors assume that the behavior of the system is modeled in a state machine notation. Therefore, the hazards have to be transformed to a statechart diagram. Their transformation of hazards is done according to the following steps:

- Identifying the types of primary events in the fault tree related to the behavioral model of the system. These types are categorized into four groups: (1) state and entry and doActivity of a state, (2) exit of state, (3) transitions, events, guard conditions and actions, and (4) data comparatives. Elements in the same category have the same transformation rules.
- developing the rules to represent the primary events and gates in a UML statechart notation, and
- extracting the information that deals with the hierarchy and the orthogonality from the original behavioral model. The output of these steps is also a state machine diagram that can be used for safety analysis.

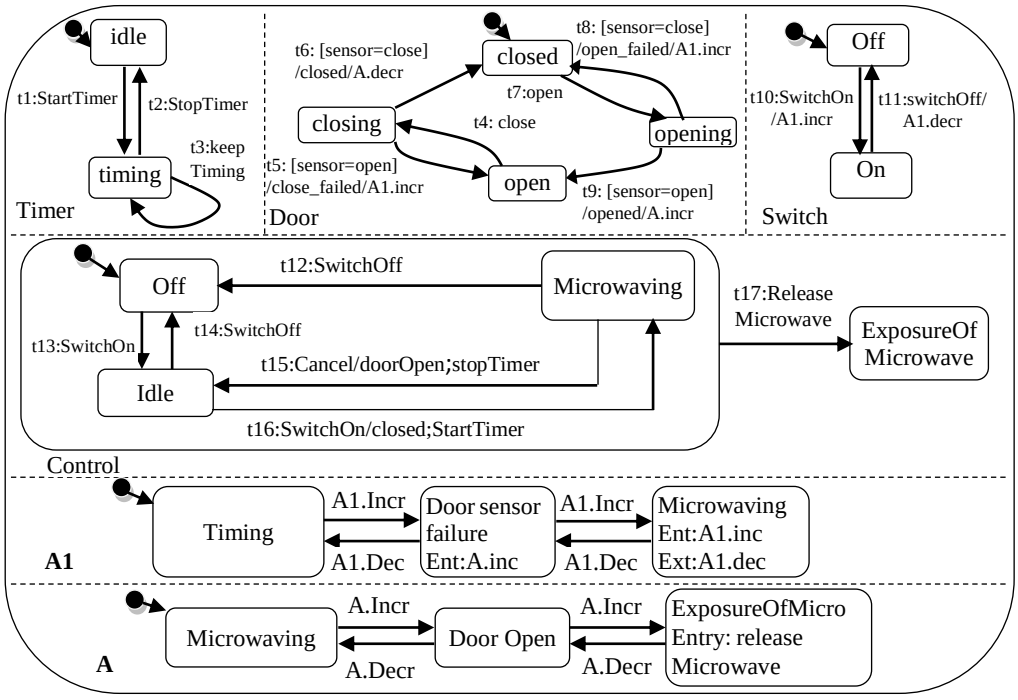

Figure 2.9: Modified Statechart for the Microwave Oven

We will use Example 2.4.1 to illustrate this technique. The first step is to identify the types of primary events in fault tree and to match them to one of doactivity in the state machine. Figure 2.10 illustrates the FTA for the microwave. This FTA is then transformed to a statechart according to the transformation rules. The AND gate in a FTA is represented as an orthogonal region in the statechart. That means both events have to happen in order for the gate to occur. Figure 2.11 shows the statechart equivalent of Figure 2.10.

After the transformation of the FTA is completed, the transition information is retrieved. Figure 2.12 shows the transformed statechart with information. This approach integrates a fault tree into a statechart model according to a set of transformation and integration rules. However, the integrated model can be used for safety analysis only. Our goal is to integrate a behavioral and fault models for safety testing.

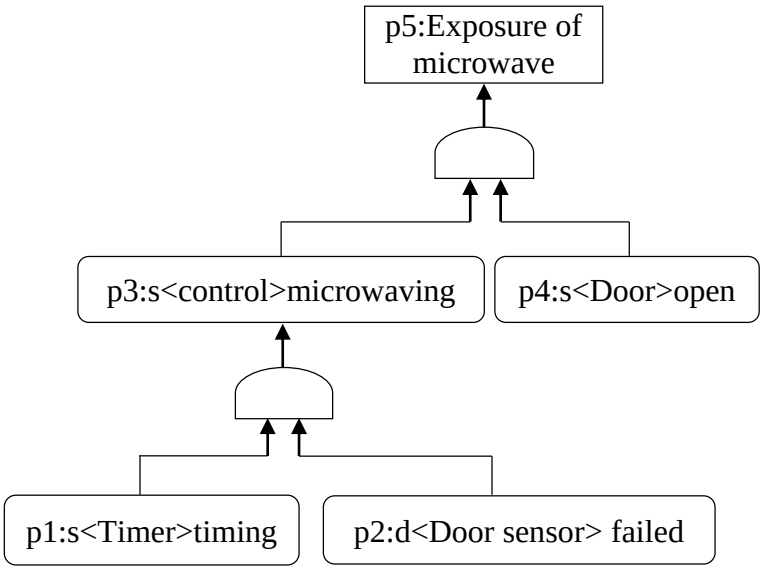

Figure 2.10: Fault Tree for Exposure of Microwave

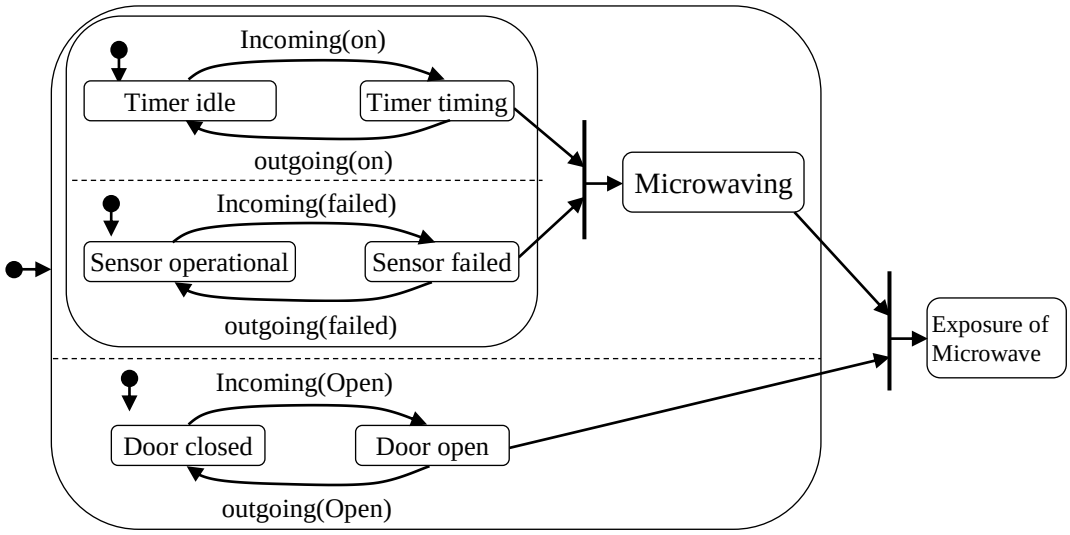

Figure 2.11: Transformed Statechart Diagram without Information from the Original Statechart diagram

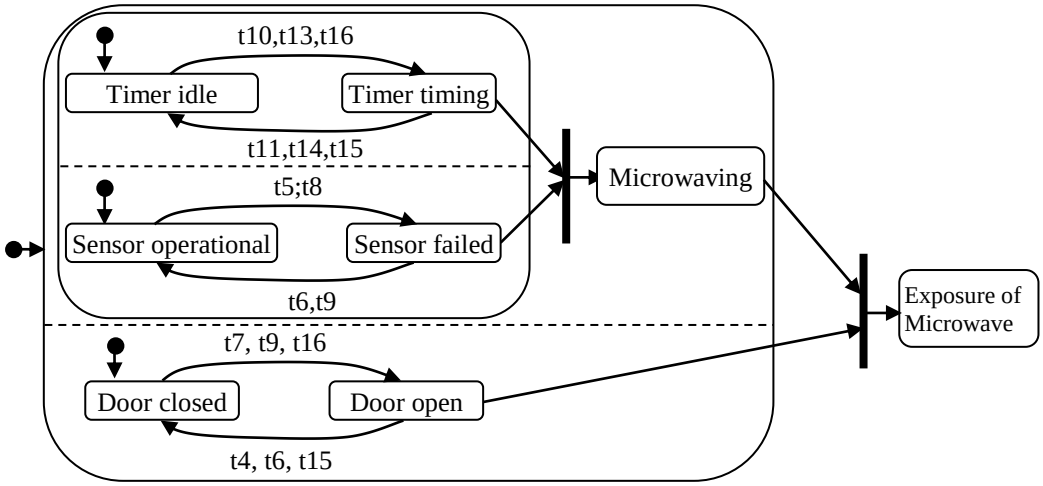

Figure 2.12: Transformed Statechart Diagram with Information

Because, usually, the system specification does not thoroughly describe the undesired behavior, Sánchez *et al.* in [127, 126] propose generating test cases based on a fault-based approach. This approach is meant to overcome the limitations of specification-based approaches that derive from the incompleteness of the specification with respect to undesirable behavior, and from the focus of specifications on the desired behavior, rather than potential faults. FTA is used to determine how undesirable states can occur in a system. The results of the analysis expressed in terms of Duration Calculus are integrated with statechart based specifications. The statechart diagram is then transformed to Extended Finite State Machines (EFSMs) to flatten the hierarchical and concurrent structure of states and to eliminate broadcast communication. Control flow is then identified in terms of the paths in the EFSMs.

Again, Example 2.4.1 will be used to illustrate this technique. The step is to find the set of basic events that can contribute to a failure. From the given FTA, the cut set for the event "Exposure of microwave" is

$$c = \lceil timing \wedge Door\ sensor\ failure \wedge Door\ open \rceil$$

The FTA node "Exposure of microwave" refers to the statechart component "control" and the sub-tree rooted "Door open" refers to the Door component, "timing" refers to the "Timer" component, and "Door sensor failure" refers to hardware component sensor. The formula denotes that the microwave is switched on, and microwave oven door is open, and the door sensors failed to operate properly. According to the transformation rules, the formula will be as follows:

$$c = \lceil timing \rceil \wedge \lceil Door\ sensor\ failure \rceil \wedge \lceil Door\ open \rceil$$

According to **rule I2 c i** in [127] that says if the failure event or state is already represented in the behavioral model, do nothing. Hence, the state "Switched on", the event "Door sensor failure" and the "Door open" state already exist in the behavioral model, there is nothing to be done.

All the aforementioned integrating techniques integrate Fault Trees with Statecharts. However, these approaches used different integration rules. We compare these techniques based on the models integrated, the use for the integrated model, and the number of states and transition in the resulting model as seen in Table 2.5. The use of these approaches is mainly safety analysis, however, the approach in Sánchez *et al.* [127] is used for safety testing after translating the integrated model, which is a statechart, into an EFSM.

Ortmeier et al. [112] present a systematic approach to formally model failure modes. The approach is combined with most formal safety analysis. They provide construction rules that ensure preserving that the initial functional behavior. They apply their method to a radio-based railroad crossing modeled using statechart. After they construct the model of the intended behavior of the system, they extend it to capture failure modes. During the extension, they split the failures into models of occurrence patterns and of direct effects modes. This allows to uniformly model a large class of occurrence patterns of failure modes (like transient, persistent etc).

The occurrence pattern of a failure mode describes under what situations a failure mode occurs. Two common patterns are transient which can appear and disappears and persistent patterns failure which when it occurs it stays forever. Deductive cause-consequence analysis (DCCA) is integrated into the presented failure model to find the minimal critical sets if failure modes.

Kaiser *et al.* [76] proposed a combination of fault trees with an explicit State/Event semantics, using a graphical notation called State/Event Fault Trees (SEFTs). This model uses the fault tree to represent the faults which are connected to the state or event in the state/event model that describes the system behavior. However, this model is used for safety analysis. Furthermore, identifying the events for an FT and connecting them to state or event is done manually which makes the process of constructing SEFT very difficult, time consuming, and error-prone especially for large and complicated systems.

Similarly, Nazier *et al.*[105] transform fault tree events into elements of a statechart behavior model. The resulting risk-based test model is used for automated test case generation by building Timed Computation Tree Logic (TCTL) queries to verify the system correctness and criticality using model checker. However, it is not clear how any coverage criteria can be imposed for interactions between orthogonal regions of the cut sets.

Table 2.5: Comparison of the Integration Techniques

Technique	FM	BM	Use	Example 2.4.1 Microwave System	
				States	Transitions
[41, 40]	FT	Statechart	Analysis	19	25
[82]	FT	Statechart	Analysis	22	28
[127]	FT	Statechart	Testing	11	16

2.4.1 Limitations

- **The limitation introduced by the gap between the models:** Combining a behavioral model and a safety model is doable in case both models are compatible. By compatible we mean that both models are described at the same level of detail, the same events in both models have the same names and attributes, and both models are dynamic. To describe the same level of detail, both models have to be at the same level of abstraction. Figure 2.13 shows where the behavioral models and fault models are used in the v-model. Although most of the models fall into the design phases of the v-model,

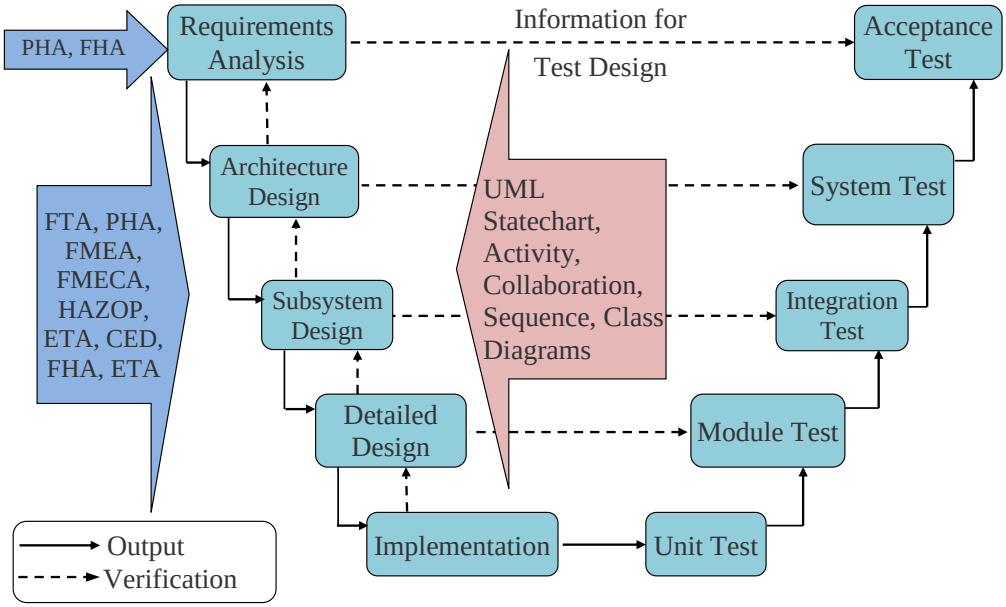

Figure 2.13: Fault and Behavioral Models at the V-model

they can be at different levels of these phases and describe different levels of abstractions, so that they cannot be compatible. Therefore, the output of the safety analysis techniques at one level will not be useful for testing at a different level. Besides that, these models can be integrated together to be

used for testing. For example, an FTA at the Architecture design level cannot be combined with a statechart at the detailed design level.

In other cases where the behavioral model describes the same level of detail as the fault model, both models have to describe the same function, component, or system (from a desired and undesired behaviors view) and both models have to have the same event names and attributes. However, in case they do not, we need to transform one model to a form that is compatible with the other. In other words, we may model safety using a modeling language that is used for modeling behavior. Doing so, we can easily integrate them in one model that considers safety aspects of the system besides the behavioral ones.

- **Scalability:** Some of the proposed model integration approaches have scalability and complexity limitations. Using a UML statechart to model fault tree gates may not be suitable because of the state explosion problem. FTA may contain hundreds of gates which means there will be hundreds of orthogonal regions since each gate is represented as an independent region inside the statechart that represents the system under test.

- **Complexity:** The transformed fault tree falls into the lowest level of the composed statechart that makes it difficult to manually or formally analyze the diagram for safety because of the indirect paths to causes of hazards [82]. These indirect paths make it difficult to generate test paths from the statechart model and make it impossible to inject faults into the system, i.e. to simulate the environment, during requirement validation.

- **No explicit mitigation models:** All hazard mitigations are implicit within the behavioral models of safety-critical systems. There are no explicit mitigation models in the form of exception handling patterns, such as emergency

stop, return to a safe state, insert an additional behavior, or try an alternative behavior.

In our approach, we will overcome these limitation by using CEFSM to model both behavioral and failure processes and how they interact. CEFSM is scalable since we can use it to model bigger systems. As for the complexity, the system modeled with a statechart may be composed of many levels of hierarchy. The hierarchical diagram can have indirect paths to causes of the hazards due to its depth (i.e., a composite state can own its sub-states) and orthogonality (i.e., regions in a state are independent of each other). Therefore, we chose to use CEFSM as a modeling language because we can keep the model flat to get rid of the system composition that is the source of complexity.

Chapter 3

Approach

In this approach, we propose an integration of the behavioral model with a fault model to take advantage of the two in the analysis and testing activities. From the testing point of view we need to:

- generate behavioral tests to test the system behavior,
- generate failures at appropriate points e.g. by injecting events into the test model [144], and
- test proper mitigation.

We also need to define coverage criteria for all three.

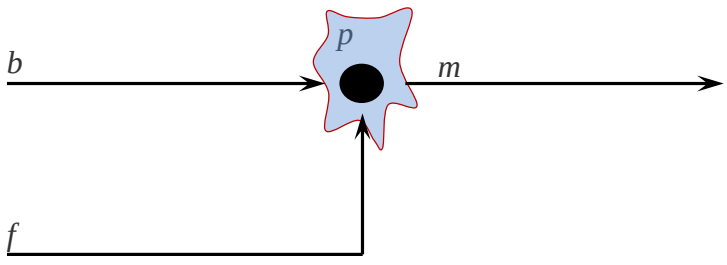

Figure 3.1: Safety-Critical System Behavior

3.1 Test Generation Process

The test generation process shown in Figure 3.2 uses the behavioral model and a FT to generate test cases. It starts with the compatibility transformation step. The FT′ produced from this step is transformed to gate CEFSMs (GCEFSMs) according to the transformation rules. Then, the model integration step integrates the GCEFSM with the behavioral model (BM) according to the integration rules. The resultant model is the Integrated Communicating Extended Finite State Machine (ICEFSM). Test case generation methods can use this model to generate test cases

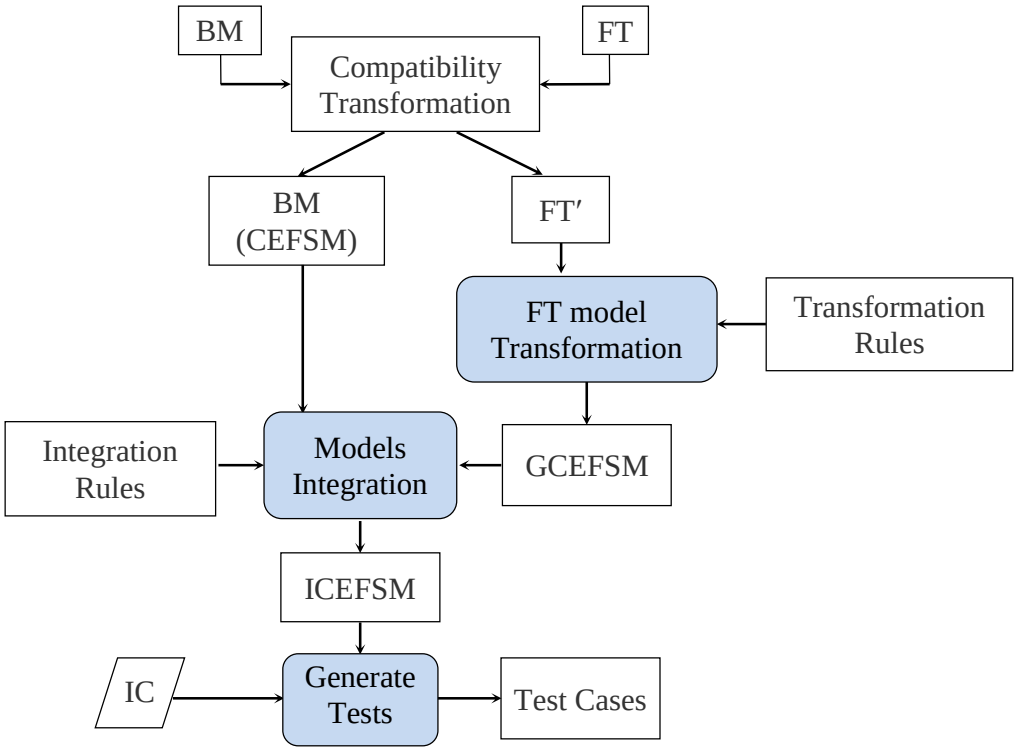

Figure 3.2: Test Process

based on test criteria (IC). Because we consider both behavior and failure occurrence to be parallel, communicating processes (*cf.*Figure 3.1), we use a communicating ex-

tended finite state machine to model their interaction. The following subsections explain each step in more detail.

At the analysis stage of the system, safety analysts will have a list of every possible failure and its ID. This list will be filled into the Failure Types Table shown in Table 3.1 at the analysis stage. This table will be used to connect phase1 (our approach) to phase2 (Generate Safety Mitigation Tests approach) of the end-to-end testing methodology. At the beginning of phase1, the first two columns contain *Failure ID* and *Failure Type* for every possible failure in the system. The following columns, *Node in FT*′, *Event ID*, *Gate ID*, and *Message ID* will be filled in during the various steps of phase1. That is, *Node in the FT*′ column is filled in during the compatibility transformation step (section 3.1.3), *Event IDs* and *Gate IDs* are filled in during the transformation procedure (section 3.1.6), *Message IDs* are filled in during the integration procedure (section 3.1.7), and *Path IDs* is filled in during test generation step (section 3.1.10).

Table 3.1: Failure Types Table Example

Failure ID	Failure Type	Node in FT′	Event ID	Gate ID	Message ID	Path ID
ex. f_1	*Gas leak*	*FBGasLeak.BFEventCond*	e_1	G_1	m_1	r_1

3.1.1 Behavioral Model: Communicating Extended Finite State Machine (CEFSM)

In principle, finite state machines can appropriately model control portions of communicating components of a system. However, practically, the usual specifications of these components include operations based on variable values; ordinary FSMs do not have the capabilities to model such systems in a concise way [88]. They

cannot model the manipulation of variables conveniently or model the exchange of arbitrary values between components.

To solve the variable manipulation problem, FSMs were extended by adding other elements used for data flow representation such as predicates, variables, and instructions, to a more advanced model called an Extended Finite State Machine (EFSM). The transfer of values between components is handled by adding communication capabilities to the EFSMs. CEFSM has been used in modeling and testing distributed systems and network protocols [15]. The strength of CEFSM is that it can model orthogonal states of a system in a flat manner and does not need to compose the whole system in one state as in statecharts which would make it more complicated and harder to analyze and/or test. The communication capabilities of the CEFSM make it suitable for modeling communicating processes.

3.1.2 Fault model: Fault Tree (FT)

Many safety analysis technique has been used to analyze. These techniques aid in the detection of the safety flaws and the design error. From these techniques, we select the fault tree to be used as our fault model to be integrated in the behavioral model in order to be used for safety testing. Fault tree is an analysis model that describes how events and failures contribute in a hazard. This is very important for our integrating approach because we need to know when and how that events and failures of the system combine to cause a hazard at the system execution.

3.1.3 Compatibility Transformation

The basic events in fault trees (leaf nodes) depend on the scope, resolution and the ground rules [143]. The scope of the FT indicates which failure will be included

and which will not, the resolution is the level of detail at which these basic events will be developed, and the ground rules include the procedure and terminology used to name these events. Often, the basic events in fault trees are informally described, i.e. in a natural language. If the resolution or the event naming does not match that of the behavioral model, which is often the case, we say these models are not compatible. Therefore, we need to make these models compatible in order to be able to integrate them. Behavioral and Fault models are said to be compatible if they describe the same level of abstraction and the same events in both models have the same meaning.

The compatibility transformation procedure takes the BM and the FT as inputs and produces a FT′ that is compatible with the BM. The attributes of entities in FT (each leaf) and behavioral model are formalized by creating a class diagram.

1. Identify entities that have capability of failure or contributing to a failure (leaf nodes). An entity could be a state or an event.
2. For each such entity, create a *Bclass* with behavioral attributes and *Fclass* with attributes related to failure and failure condition.
3. Express entity.failure condition in terms of attributes of *Fclass*.
4. Combine both *Bclass* and *Fclass* by identifying attributes common to both diagrams such that, if values in *Fclass* and *Bclass* are the same, we combine the attributes, otherwise we create *Battribute* and *Fattribute*.

Figure 3.3 shows a *BClass*, a *FClass*, and a *BFClass*. The *BClass* contains either a state B_S (a state at the behavioral model) or an event B_E (an event at the behavioral model) from the behavioral model that contributes to a failure at the fault model. These events are carried in the communicating messages from the behavioral to the fault models when these models are integrated. The *FClass*

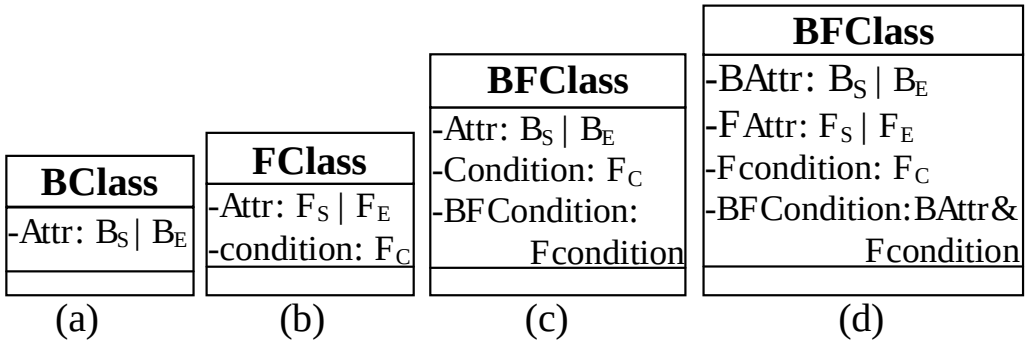

Figure 3.3: Behavioral and Fault Classes Combination

contains a state F_S (a state at the failure model) or an event F_E (an event at the failure model) as described in a leaf node of a FT along with their conditions F_C (if any). The *BFClass* contains either a combination of *BClass* and *FClass* attributes if these attributes are the same as shown in Figure 3.3 (c) or separate *Battributes* and *Fattributes* are created as shown in Figure 3.3 (d). At this step, the column *Node in FT′* in the Failure Type Table is filled in with the related leaf node from the FT′ for every *Failure ID*.

For example, the fault tree ($\wedge$,*Air Present*, *Gas leaks > 4 sec*) depicted in Figure 3.4 contains two events that contribute to an unsafe environment. These events need to be made compatible with the events that have the same meaning in the behavioral model. Let us assume that the entities that have capability of failure or contributing to a failure in the behavioral model are "Air Valve" and "Gas Valve". Therefore, *BClass*, *BAirValve*, will be created for the entity "Air Valve". The attribute of this class is of type B_S and its values are "Open" or "Closed". The *FClass*, *FAir-Valve*, will be created for the leaf node "Air present". The name of the attribute is "AirPresent", its type is F_S, its values are "yes" or "no", and the condition of this attribute *AirPresent = yes*. Since the names of these entities are not the same although they have the same meaning, we create a *BFAirValve* class that contains

separate attributes of both $BAirValve$ and $FAirValve$ as described in Figure 3.3 (d). The *BFAirValve* attributes are *BState:Open, Closed*, *FState:Airpresent:yes, no* and *BFEventCondition:AirPresent= yes* (Figure. 3.5). Also, the $Bclass$, *BGas-Valve*, will be created for the entity "Gas Valve". The attribute of this class is of type B_S and its values are "Open" or "Closed". The $FClass$, *FGasValve*, will also be created for the leaf node "Gas leaks > 4 sec". The name of the attribute is "Leaks", its type is F_S, its values are "yes" or "no", and the condition of this attribute *Leaks & TimeInState* > 4 *sec*. We create a *BFGasValve* class that contains separate attributes of both $BGasValve$ and $FGasValve$. The *BFGasValve* attributes are *BState:Open, Closed*, *FState:Leaks:yes, no*, *FTimeInState:4 sec*, and *BFEventCondition:Leaks & FTimeInState > 4 sec* (*cf.* Figure 3.6).

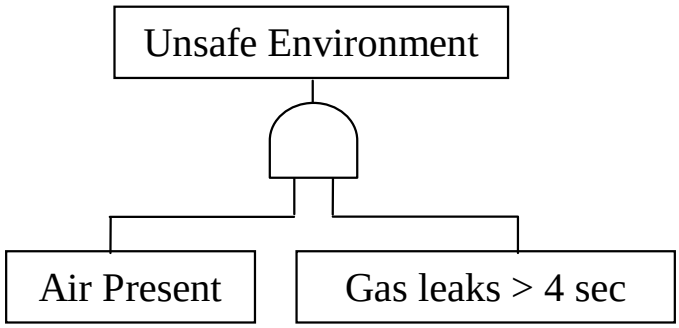

Figure 3.4: Fault Tree Example

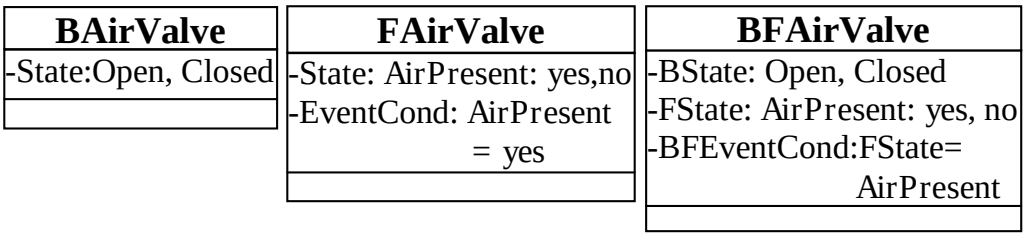

Figure 3.5: Air Valve Class

At this point, the conditions are aggregated from the leaves of the FT to the root. The compatibility transformation is an essential step to solve the ambiguity

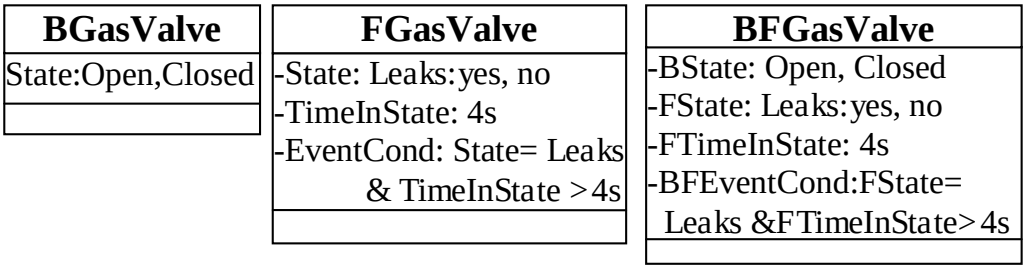

Figure 3.6: Gas Valve Class

between the events in the behavioral model and fault model. The output of this step is a FT' which is described in terms of *BFClass.BFEventCondition* combined with logical operators. The compatible fault tree for this example will be: $FT' = (\wedge, BFAirValve.BFEventCond,\ BFGasValve.BFEventCond)$

3.1.4 FT′ model Transformation

Events can be classified as either "transient" or "persistent" [111]. A transient event is an event that is reversible i.e. it can appear and disappear completely, while the persistent event once it occurs, stays. An ordinary fault tree, which statically describes hazard, does not consider this distinction between events because this distinction would not make a difference for a static model. However, it is essential to consider the event type attribute when making a fault tree dynamic. The event type determines if the status of the event can be "not-occurred" after it had already "occurred". The change of the event status makes the integrated fault tree react according to the status of the event in the behavioral model. Note that our transformation rules allow for modeling transient events unlike the classical fault trees where all the events are persistent.

However, the FT′ is a static model that describes the hazard as a specific combination of events. In order for the FT′ to be integrated into a behavioral model it

has to be dynamic and understands the behavioral model's events. To accomplish that, we have to transform the FT′ to a CEFSM format. Every gate in the FT′ is represented as a GCEFSM. The whole model forms a tree-like structure. The ICEFSM consists of a collection of CEFSMs that represent the behavioral model and GCEFSMs (the transformed FT) model.

The communication between the behavioral model and FT′ model is achieved by sending and receiving messages between the models. The behavioral model sends messages to the Fault related GCEFSMs, but they do not send any message back to the behavioral model. Upon receiving those messages, the GCEFSMs at the lower level of the tree sends messages that carry "the event occurred" or "has not occurred" to the upper level GCEFSMs and so on. The output message from one GCEFSM is taken as a parameter to a generic event in the receiving GCEFSM, *e.g.* $event(param) = get(m_i)$.

To make the FT′ to GCEFSM transformation automatic, the representation of the FT events and gates in GCEFSM is standardized. Each gate must be given an identifier that uniquely identifies it. The output of the gate, which is an input to another gate, should carry the same identification number as the gate that outputs it. If the gate event has occurred, a message m_i is sent to the receiving gate indicating that the event has occurred. The output of each gate is an input to another gate. The GCEFSM may be in one of three conditions; it has not received any input messages so far, it received a message that says the gate event has occurred, or received a message that says the gate event has not occurred.

3.1.5 Transformation Rules

The transformation rules use the notation for *e,m* introduced for CEFSM in section 3.1.1. Every gate in the FT′ is converted to an equivalent representation in CEFSM i.e. Gate CEFSM (GCEFSM). Every GCEFSM is identified by a unique identifier G_i that uniquely identifies the gate. The set of variables V are:

- *TotalNoOfEvents* is the total number of input events to the gate.
- *NoOfOccurredEvents* is the number of occurred events that the gate received so far.
- *NoOfPositiveEvents* is the number of occurred events whose *eStatus* is true.

Each GCEFSM consists of states and transitions that perform the same boolean function as the gates in an FT. The difference is that in the original FT, a gate produces a single output when all the input events satisfy the gate conditions. Otherwise, no output would be produced. However, in the transformed FT, a gate has two kinds of outputs. One output is defined as the "Gate occurred" and the other is defined as "Gate not occurred" such that:

$$m_i = \begin{cases} \text{Gate Occurred} & \text{if } G_i(e_1, e_2, ...e_k) = true, \\ \text{Gate not Occurred} & \text{if } G_i(e_1, e_2, ...e_k) = false \\ & \quad and\ eOccurrence = true \\ & \quad \forall e_i, i = 1\ to\ k \end{cases}$$

For example, an AND gate = true if $G_{AND}(e_1 \cap e_2 ... \cap e_k) = true$. Each structure and behavior of each GCEFSM is predefined and for this matter we will present each gate as follows:

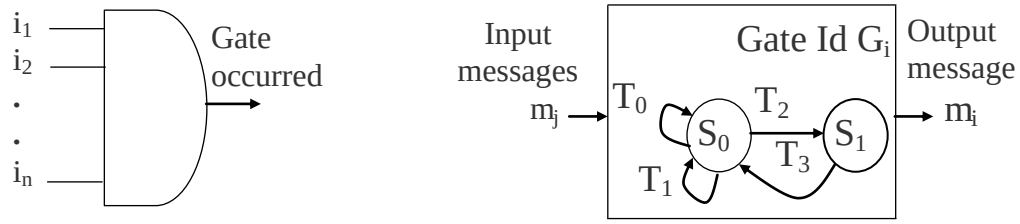

Figure 3.7: AND Gate Representation in FT and GCEFSM

3.1.5.1 AND Gate

When combining some events with an AND gate, the output occurs when all the events occur. Otherwise, no output would occur. An AND gate is represented as shown in Figure 3.7. It consists of two states and four transitions. State S_0 is the initial state and S_1 is the "gate occurred" state. The transition T_2 will never be taken unless its predicate *NoOfOccurredEvents=TotalNoOfEvents & e_i.eOccurrence =true & e_i.eStatus = true* is true which means all the inputs are received and their status is true. When T_2 is taken the message "gate occurred" is sent to a GCEFSM that is supposed to receive it. The transition T_0 is as follows:

T_0:(S_0, [e_j.eOccurrence = true & e_j.eStatus = true & NoOfOccurredEvents < TotalNoOfEvents], get(m_j))/ (S_0, update(events),-) Where,

1. The event get(m_j) obtains input messages from the environment or from another CEFSM. m_i contains an event that could be "gate occurred" or "gate not occurred".

2. update(events) is an action performed upon the executing of this transition. It updates the number of occurred events and their status based on the last input message received.

3. The predicate "[e_j.eOccurrence = true & e_j.eStatus = true & NoOfOccurredEvents < TotalNoOfEvents]" ensures that the event has occurred and the

number of inputs received so far is less than the total number of inputs and the input status is true. Note that "gate not occurred" implies that *eOccurrence=true&eStatus=false*, while "gate occurred" implies that *eOccurrence=true&eStatus=true*.

If all the messages to this GCEFSM are received and all the events have occurred, the transition T_2 will be taken.

T_2*:(S_0,[NoOfPositiveEvents=TotalNoOfEvents &* e_j*.eOccurrence=true &* e_j*.eStatus = true], get(m_j))/(S_1, update(events), Send(Gate Occurred))*

When this transition is taken based on the input and the predicate, it moves to state S_1, increments the number of inputs, and send an output message saying that the gate has occurred.

T_1*: (S_0,[e_j.eOccurrence = true &* e_j*.eStatus = false], get(m_j))/(S_0, update (events),-)*, where "-" means no output produced.

When on state S_0 and the input message implies that the event has changed its status, the transition T_1 is taken. T_1 decrements the number of inputs, and updates the status of the event from occurred to not occurred.

T_3*:(S_1, [e_j.eStatus = false],get(m_j))/(S_0, update(events), Send(Gate not Occurred))*

At the state S_1, Transition T_3 is taken when the coming input status is false. When this transition is taken it decrements *NoOfOccurredEvents* and *NoOfPositiveEvents*, updates the status of the input from occurred to not occurred and sends "gate not occurred" message to the receiving gate.

3.1.5.2 INHIBIT Gate

INHIBIT is similar to the AND gate. They have the same states and transitions. The only difference is that the predicate for the transitions T_2 and T_3 should include

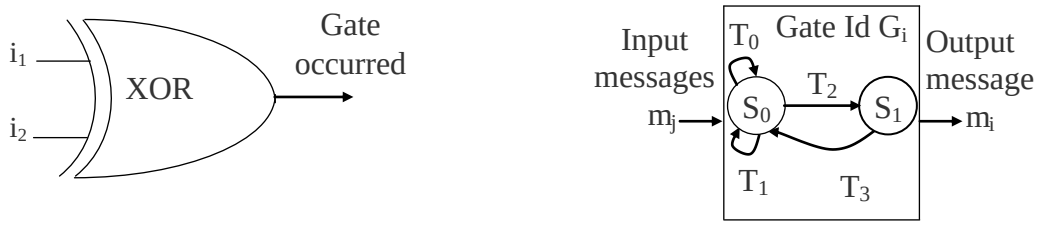

Figure 3.8: XOR Gate Representation in FT and GCEFSM

the enabling condition. We do not need to have a separate gate representation for NOT gate since we can express it in any predicate. If we want to negate any event we can use the NOT logical operator inside the gate that the negated event is one of its inputs.

3.1.5.3 XOR Gate

This gate is slightly different from the AND gate although it has the same structure and same number of transitions and states. At this gate, it is necessary to distinguish between the event that has not occurred in the first place and the one whose status is false. The representation of GCEFSM XOR gate is shown in Figure 3.8. T_0 to T_3 are the possible transitions that may be taken based on their predicates.

T_0:(S_0,[NoOfOccurredEvents=0&e_j.eOccurrence=true], get(m_j))/(S_0, update (events))

T_1:(S_0,[NoOfOccurredEvents=1 & e_j.eOccurrence= true & xor(events)=false], get(m_j))/(S_0,update(events),-)

T_2:(S_0,[NoOfOccurredEvents=1&e_j.eOccurrence=true & xor(events)=true], get(m_j))/(S_1,update(events),Send(gate occurred))

T_3:(S_1,[inputStatusChanged(e_j)=true],get(m_j))/(S_0, update(events),Send(gate not occurred))

3.1.5.4 Priority AND Gate

As seen in Figure 3.9, the priority AND gate is very similar to the AND gate in the overall structure and transitions. It differs from the AND gate in that the events have to happen in predetermined order. This difference is taken care of by manipulating the predicate condition in such a way that it considers the order of occurrence of the events. For example, if the events are ordered E_0 then E_1, they have to happen in this order so that the gate can occur. Otherwise the gate will not occur. T_0 to T_3 are the transitions that control the priority AND gate.

T_0*:(S_0,[NoOfPositiveEvents< TotalNoOfEvents & e_i.eStatus= true],get(m_i))*
/(S_0,update(events),-)

T_1*:(S_0,[e_j.eStatus = false & e_j.eOccurrence = true], get(m_j))/(S_0,*
update(events),-)

T_2*:(S_0,[NoOfPositiveEvents = TotalNoOfEvents & ordered(e_j) = true &*
e_j.eStatus = true],get(m_j))/(S_1,update(events), Send(Gate Occurred))

The predicate *ordered()* returns true of the input event in the message m_i is received in its predefined order and returns false otherwise.

T_3*:(S_1, [e_i.eStatus = false], get(m_i))/(S_0,update(events), Send(Gate Not Occurred))*

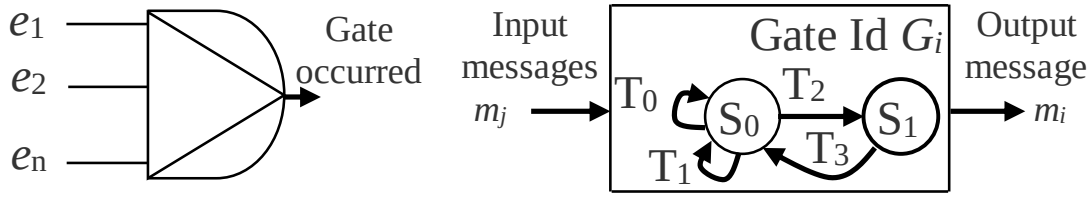

Figure 3.9: Priority And Gate Representation in FT and GCEFSM

3.1.5.5 OR Gate

The OR gate occurs if at least one event occurs. This gate, as seen in Figure 3.10, consists of two states and four transitions. When in S_0 and the input message carries an event whose *eOccurrence* and *eStatus* are true (i.e. the event has occurred), T_0 is taken and the OR gate occurs. In state S_1 and if the events in the input messages have not occurred (i.e. their *eStatus* is false) and there was only one input so far, which means this input has changed its status, then a "Gate not occurred" message is sent. Otherwise, no message is sent out of this gate and only update(events) actions take place.

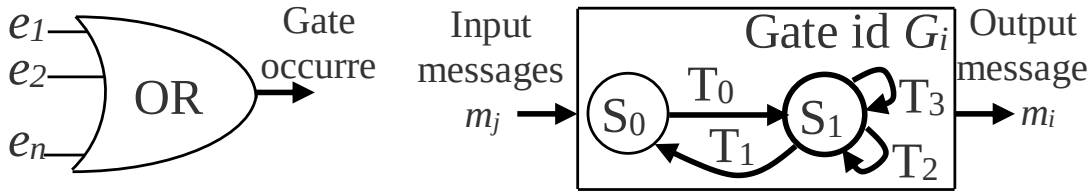

Figure 3.10: OR Gate Representation in FT and GCEFSM

T_0:(S_0,[e_j.eStatus=true],get(m_j))/(S_1,update(inputs), Send(Gate Occurred))

T_1:(S_1,[e_j.eStatus = false & NoOfPositiveEvents = 0], get(m_j))/(S_1, update (inputs), Send(Gate not Occurred))

T_2:(S_1,[e_j.eStatus= true],get(m_j))/(S_1,update(events),-)

T_3:(S_1,[e_j.eStatus= false],get(m_j))/(S_1,update(events),-)

3.1.5.6 Timing an Event Gate

FT gates such as AND, OR, INHIBIT, etc. are well defined and can be syntactically represented. Events in FT can be simple or composed. A composed event can be decomposed further to simple events or a timed simple event. A timed simple

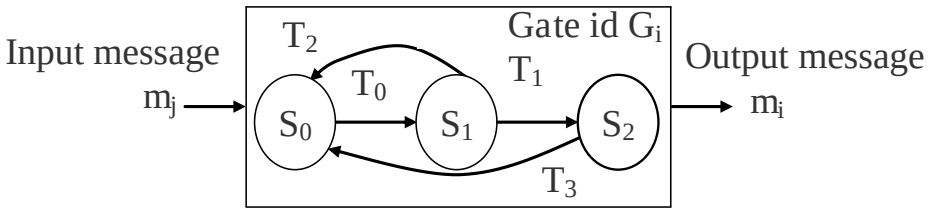

Figure 3.11: Event Timer GCEFSM

event is the simple event that should occur for a specific period of time to contribute to a hazard. However, FT has no timing gates. Therefore, we need to have a representation that can handle the timing issue (either a minimum or maximum timing).

Thus, we introduce this gate that can time an event and the gate in the subsection 3.1.5 that deals with the timing intervals. This gate works as follows. Upon receiving a message that indicates the occurrence of the event, the transition T_0 takes place which starts the timer. When the time expires and no further "gate not occurred" message was received that indicates that the event is no longer happening, the transition T_2 is taken and sends a "gate occurred" message. Otherwise the gate does not occur. T_2 is taken when the event status e_i.eStatus changes to false.

T_0*:(S_0,[e_j.eStatus=true],get(m_j))/(S_1,setTimer(v,$Timer_i$),-)*

T_1*:(S_1,timeout)/(S_2,-,Send(GateOccurred))*

T_2*:(S_1,e_j.eStatus= false],get(m_j))/(S_0,reset($Timer_i$);updat e(events))*

T_3*:(S_2,e_i.eStatus= false],get(m_i))/(S_0,reset($Timer_i$);update(events),Send (GateNotOccurred))*

3.1.5.7 Timing an Event for Continuous Intervals Gate

Some event may need to be timed for continuous intervals. For example, we may need to observe an occurrence of an event every consecutive 5 sec as long as the system is operational. Figure 3.12 shows that as long as the transition T_0 is fired and T_2 was not, the event will be timed for fixed consecutive amount of time and it keeps timing until the status of the event $e_i.eStatus$ changes to false. Upon receiving this event change, the transition T_3 to the state S_1 is fired sending out a "gate not occurred" message.

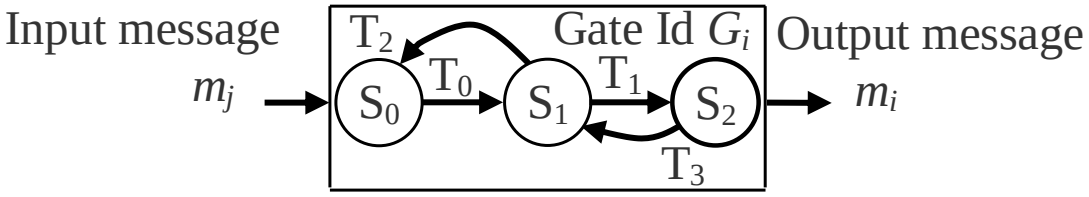

Figure 3.12: Timing Continuous Intervals GCEFSM

T_0:(S_0,[$e_j.eStatus = true$], $get(m_j)$)/(S_1, $setTimer(v, Timer_i)$, $Send(Gate\ Occurred)$)

T_1:(S_1,$timeout$)/(S_2,-, $Send(Gate\ not\ Occurred)$)

T_2:(S_1,[$e_j.eStatus = false$],$get(m_j)$)/(S_0,$reset(Timer_i)$, $Send(Gate\ not\ Occurred)$)

T_3:(S_2,$setTimer(v,Timer_i)$)/(S_1,-,$Send(Gate\ Occurred)$)

3.1.6 Transformation Procedure

As mentioned above, the transformed GCEFSMs form a tree-like structure and each GCEFSM gate is denoted by a unique identifier G_i that uniquely identifies the gate. The transformation procedure shown in Figure 3.13 takes an FT as an input and produces GCEFSMs according to a postorder tree traversal. Event-gate table is used for the integration of GCEFSMs model with the behavioral model. It contains

```
Procedure FT_TO_GCEFSM (T : Tree)
{
  if (T is null)  then return;
  for each child C of T from left to right do
     FT_TO_GCEFSM(C);
     Construct GCEFSM gate;              // Create a gate with its variables,
                                         //  output messages, and its ID.
   if (leaf node)
    then   insert event name, event ID & Gate ID into Event-Gate   table.
}
```

Figure 3.13: Transformation Procedure

the entries for all leaf nodes of the FT and is defined as shown in Table 3.2. This table is constructed during the transformation of FT to GCEFSM. The leaf node event name and identifier are inserted into the table entry along with the identifier of the gate that receives this event. At this step, the columns *Event ID* and *Gate ID* in the Failure Type Table are also filled in with the event id and the gate id for every *Failure ID*.

Table 3.2: Event-Gate Table for Leaf Nodes

Event name & attribute	Event ID	Gate ID
event name as indicated in the FT	e_i, where $(i = 1, ..., n)$ and e_i is leaf connected to G_j	G_j
$Ex.\ temp > 10\,^{\circ}\mathrm{C}$	e_1	G_1

3.1.7 Integration Procedure

Before integrating the models, all the messages from the behavioral model to the fault model have the form of equation (3.1.1). At that time the *event id* contains

the events name and attribute and the receiving gate *id* of that event is not known yet. During the integration of both models, the event name in each message in the behavioral model is looked up in the event-gate table. If the event name and attribute in the behavioral model match those in the event-gate table, the message is modified such that it contains the *event id* e_i and gate *id* G_j as stated in equation (3.1.2) according to the procedure in Figure 3.14.

```
Procedure ModelsIntegration(BM,Event-Gate Table){
 For every mBk Do
  For every Event-Gate entries Do
   If(mBk. EventNameAndAttribute == Event-Gate.EventNameAndAttribute) then
            mBk.EventID       = Event-Gate.ei
            mBk.mDestination = Event-Gate.Gi
}
```

Figure 3.14: Integration Procedure

$$Let\ m_{Bk}\ =\ (mId, EventNameAndAttribute, \_)$$

$$be\ a\ message\ from\ the\ BM \tag{3.1.1}$$

$$m_{Bk}\ will\ be\ modified\ to\ (mId, e_j, G_i) \tag{3.1.2}$$

The column *message ID* in the Failure Type Table is filled in with the message id that carries the event id for every *Failure ID*.

3.1.8 Concurrent Processes

Our integrated model consists of communicating behavioral and failure processes. These processes have their internal paths that exhibit the execution of task paths

for each individual $CEFSM_i$ and the communicating messages that show the global paths that represent the communications between the CEFSMs during their executions. The behavior varies according to the change of the synchronized condition among the concurrent processes. Therefore, we need to consider the concurrent paths of the model. The concurrent path is generated by the Cartesian product of CEFSMs paths. Each produced concurrent path is an arbitrary combination of CEFSMs paths and not always a real concurrent path. Thus, a huge number of paths is produced and therefore, different concurrent path representation approaches were introduced. Weiss [148] serializes the concurrent program as a set of sequential programs to produce test paths. However, this approach was criticized by Yang *et al.* in [151] as generating serialization for a concurrent programs is difficult and tedious task especially for a large number of lengthy processes. Another approach proposed by Liu *et al.* [95] uses reachability analysis to identify the concurrent paths from the whole production of the candidate concurrent test paths. However, their approach requires a large memory space and a long verification time for the concurrent paths. Therefore, for our model, it is more suitable to use the concurrent path representation used in [71, 151]. They represent the concurrent paths as ordered paths of the processes involved in the execution of a specific task and they defined the concurrent paths as follows:

Let $\mathbb{P}$ be a set of concurrent CEFSMs that consists of $CEFSM_1$, $CEFSM_2$, ...,$CEFSM_n$ where n is the number of the processes in $\mathbb{P}$.

A test path is a sequence of nodes, $n_0 n_1 n_2 ... n_m$, where n_0 is the starting node, n_m is the ending node and for each $0 \leq j < m$, $(n_j, n_{j+i}) \in E_i$, E_i is the set if edges in $CEFSM_i$. A path represents one possible execution sequence of a $CEFSM_i$.

In the execution of $\mathbb{P}$, each test case will traverse a path through one or more CEFSMs. Therefore, the execution can be seen as involving a set of paths of con-

current processes. A concurrent path is an n-tuple $(P_1, P_2, ..., P_n)$ where, for each i, P_i is a test path.

A feasible concurrent path is a path $(P_1, P_2, \ldots, P_n)$ where at least one input x causes P_i to be traversed during the execution of the system with x. The concurrent path may be infeasible due to data or communication (rendezvous) dependencies, or could be a result of the arbitrary production of paths, that is, paths that are not related.

Let p_i be the number of paths of $CEFSM_i$. Then the possible number of concurrent paths $|CP|$ of the system $\mathbb{P}$ is $\prod_{i=1}^{n}(p_i + 1) - 1$ [151].

Let $\mathbb{P}$ be a concurrent system, LG=$(LG_1, LG_2, \ldots, LG_n)$ the local view or the internal graph of the system and GG=$(GG_1, GG_2, \ldots, GG_n)$ the global view or the rendezvous graph of $\mathbb{P}$.

The length of the test suite for the whole integrated ICEFSM model is *len(CP)* states.

Let $\mathbb{L}$ be the set of paths of LG, and $\mathbb{L}_0 \subset \mathbb{L}$ be a subset of LG. Let $\mathbb{G}$ be the set of paths of GG, and $\mathbb{G}_0 \subset \mathbb{G}$ be a subset of GG.

3.1.8.1 Rendezvous Graph

To define a concurrent coverage criteria, we need to define the rendezvous graph. According to Yang *et al.* in [151], A rendezvous graph for a task T_i, $RG_i = (RE_i, RN_i)$ is obtained from the task graph of Task T_i according to the following rules:

1. Delete all nodes that do not send or receive messages except the start and the end node. An edge between n_i and n_j exists if there is a path from n_i to n_j where there is no rendezvous node between n_i and n_j.

2. If a node n_a which is involved in the communication sends or receives messages k tasks, the node is replaced by nodes (n_s,n_1,n_2,..,n_k) such that each node sends or receives one communication message and communicate with one other task from other task graph. That is, each node represents that T_i rendezvous with one of the k tasks. The edge (n_s, n_j) $\in RN_i$ for each 1 $\leq$ j$\leq$k, For each edge (n_x, n_a) $\in E_i$, there is an edges (n_x, n_s) $\in RN_i$ and for each edge (n_a,n_y) $\in E_i$, there is k edge (n_j,n_j) $\in RN_i$, 1 $\leq$ j$\leq$k.

3.1.9 ICEFSM Coverage Criteria

Exhaustive testing of all possible behaviors of a system is costly and may not be feasible. Therefore, an adequate subset of the complete testing behavior has to be selected and used in the testing process. This subset is produced according to some coverage criteria and often used to control the test generation process or to measure the quality of the test suite. A coverage criterion is usually defined regardless of any specific test model. Defining coverage criteria for the CEFSM model should consider the local view which deals with the internal states and transitions of every individual $CEFSM_i$, and the global view which considers the behavior of the whole system. Every individual CEFSM's behavior is, in fact, sequential, and therefore, coverage criteria defined for sequential programs, such as node and edge coverage can be used.

The Integrated CEFSM (ICEFSM) model is a collection of concurrent processes. Each process is modeled as a $CEFSM_i$ that can be represented as a directed graph $G_i = (N_i, E_i)$ where N_i is a set of nodes and E_i is a set of edges and is considered as a conventional graph where it is treated sequentially [151]. These CEFSMs com-

municate via the exchange of messages. It is clear that we have two different kinds of paths that represent the execution behavior of the concurrent systems.

3.1.9.1 Internal Coverage Criteria

The internal paths that describe the internal execution of the process that can be characterized by the input and the sequence of the states involved in the execution. This can be described as a static structure. The static structure, however, is not really applicable for modeling concurrent programs because the behavior of the concurrent system can not be determined by an input and a sequence of states of each individual process involved in the execution [135].

The first class of coverage criteria is the same as the graph coverage criteria defined in [4]. These coverage criteria are suitable for the internal structure of CEFSM since each CEFSM is described as a directed graph that behaves sequentially. Criteria such as Structural Coverage (Node Coverage, Edge Coverage, Edge-Pair Coverage, Prime Path Coverage, ...) or Data Flow (All-Defs Coverage, All-Uses Coverage, All-du-Paths Coverage, ...) can be used.

3.1.9.2 Concurrent Coverage Criteria

The second class of coverage criteria is defined on the global view (rendezvous) graph GG in which we consider every $CEFSM_i$ as one node without getting into the internal details. These criteria work for the global view of the integrated model, especially for the GCEFSM part, where the GCEFSM accepts more than one input to produce output. The internal details is already covered by the first criteria.

Another criteria is defined for the global view of the integrated model. This criteria is defined for the test coverage of whole ICEFSM's execution behavior, especially for the fault tree (GCEFSMs) where we already know their internal behavior and

since we are concerned with whether the events meet at a GCEFSM and whether an output from that GCEFSM is produced. Therefore, we consider each $CEFSM_i$ as a rendezvous state without going into its internal details. Based on our integrated model, we define the following two classes of coverage criteria for concurrent processes testing.

1. CEFSM coverage (CC) - Each CEFSM in the rendezvous graph is visited at least once. $\mathbb{G}_0$ satisfies CC iff for each CEFSM $c_0 \in GG$ there exists a path $G \in \mathbb{G}_0$ visits that c_0. The rendezvous node at the fault tree part of the model mean a GCEFSM.

2. Message edge coverage (EC) - Each message edge should be tested at least once. $\mathbb{L}_0$ satisfies EC iff for each edge$(c_1, c_2) \in GG$ there exists a path $G \in \mathbb{G}_0$ such that (c_1, c_2) is passed by a path along G. The message edge means every message comes in or out of a $CEFSM_i$.

3. Message path coverage (PC) - All paths of each individual test path should be visited at least once. $\mathbb{G}_0$ satisfies PC iff for each syntactic path SP $\in GG$ there exists a semantic path $G \in \mathbb{G}_0$ such that SP is passed by G.

4. Concurrent path coverage (CP) - Each concurrent path of rendezvous graph should be visited at least once. $\mathbb{G}_0$ satisfies CP iff for each syntactic concurrent path P $\in GG$ there exists a semantic path $G \in \mathbb{G}_0$ such that SP is passed by G. This criterion is meant to cover the fault tree FT part of the model's minimum cut sets.

5. GCEFSM leaves coverage (GL) - Each leaf GCEFSM at the fault model part in the rendezvous graph is visited at least once. $\mathbb{G}_0$ satisfies GL iff for each GCEFSM $c_0 \in GG$ there exists a path $G \in \mathbb{G}_0$ visits that c_0. The rendezvous node at the fault tree part of the model mean a GCEFSM.

3.1.10 Test Case Generation

A number of existing test generation methods for CEFSMs can be used. One approach to testing CEFSMs is to compose them all into one machine at once, using reachability analysis to generate test cases. However, this approach is impractical due to the state explosion problem and the presence of variables and conditional statements. Some work has been done in testing the behavior of concurrent systems and network protocols that were modeled using CEFSM. Hessel *et al.* [68] and Bourhfir *et al.* [17, 16]. They use reachability analysis to generate test cases from systems modeled in CEFSMs, while Kovas *et al.* [86] design methods and mutation operators to enable the automation of test selection in a CEFSM model. Henniger *et al.* [67] generate test purpose description of the behavior of a system of asynchronously CEFSMs. [86] use mutation to enable the automation of test selection in a CEFSM model. [15] combines specification and fault coverages to generate test cases in CEFSM models. Li *et al.* [93] proposes a methodology to generate test cases from CEFSM-based models. At this step, the column *Path ID* in the Failure Type Table is filled in with the path number that contains the failure for every *Failure ID*.

Chapter 4

Validation

In this chapter, we validate our approach by investigating scalability in section section 4.1 and applicability in section 4.2. To investigate scalability, we built a simulator that calculates the size of the integrated models (number of states and transitions) of our approach and estimates the sizes of the integrated models of Sánchez *et al.*'s [127]. We compared the result of the two approaches because Sánchez *et al.*'s approach [127] is the only approach that models integration for the purpose of test case generation and found that our approach scales better than Sánchez *et al.*'s. We varied the behavioral model and the fault model sizes to show how scalable our approach is.

To show applicability, we applied our approach on case studies from different application domains (*cf* sections 4.2.1, 4.2.2, and 4.4) hence our approach is not domain specific. Generally, we expect that it can be used for systems modeled using CEFSMs. We also used case studies that reflect different model sizes and integrated multiple fault trees to show that this approach can be used as described in chapter 3.

Applicability also means that our approach fits into an end-to-end testing methodology. Section 4.3 describes how our approach fits in an end-to-end testing method-

ology. We apply it to a case study. This work was done jointly with Salwa Elakeili. Validating the effectiveness is limited by the existing test generation techniques such as [68, 16, 86] that are being used for test generation with CEFSMs.

4.1 Scalability and Comparison to Sánchez *et. al.*'s [127]

4.1.1 Simulator Experiment Design

The simulator is intended to calculate the number of states and the transitions of the integrated behavioral and fault models according to our approach's transformation rules (CEFSMs with FTs) and Sánchez *et al.*'s approach (EFSM from statecharts and FTs) [127]. We fed the simulator with input date of different ranges of BM and FM. The behavioral models vary from 13 states and 15 transitions to 50 states and 60 transitions while the fault trees that vary from 5 leaves to 19 leaves as shown in Table 4.2. We assume that every behavioral model is integrated with every fault model. Therefore, The simulator calculates the size (states and transitions) of the integrated model of every behavioral model with every fault model as inputs.

4.1.2 Comparison of the Number of Nodes and Transitions

We developed a tool to calculate the number of states and transitions of the integrated behavioral and fault models according to the approach's transformation rules (CEFSMs with FTs) and to estimate the number of nodes and transitions according to Sánchez *et al.*'s approach (EFSM from statecharts and FTs) [127].

4.1.2.1 In CEFSM

The tool calculates the number of nodes and transition of the integrated model by adding the number of the nodes and transitions of the behavioral model to the number of the states and transitions of the GCEFSM part of the model. As we have seen in section 3 the FT gates are transformed into a collection of GCEFSMs. Every GCEFSM has a specific number of nodes and transitions. Thus, the tool calculates the number of nodes and transitions of the ICEFSM based on the number and type of gates.

4.1.2.2 For EFSM in [127]

The tool estimates the number of nodes and transitions of the integrated model according to the approach's transformation rules. The integration rules of the the approach by Sánchez *et al.* [127] use the minimum cut set of the leaf node events. For every member of the cut set they create an independent region, add a state to the behavioral model, or do nothing. This depends on whether the event already existed in the behavioral model, or may need human intervention to decide whether to model the cut as an independent region or to add it to the behavioral model as a single state and transition. Therefore, we calculated the size of the integrated model based on these options repeatedly and computed an average. Each time we change the percentage of creating an independent region. We run the tool 10 times for each input data varying the probability of creating an independent region between 50% and 60%. Then we calculated the confidence interval with a confidence level of 95%, alpha = 0.05%, for each run to show that we have taken into account the non-determinism of the estimation of the number of the states and transitions introduced

by the EFSM approach of Sánchez *et al.* [127]. Therefore, we can say that there is a 95% chance that this confidence interval contains the true population mean.

4.1.2.3 Comparison of Case Studies

We compare the number of nodes and transitions between the model integration approach presented here and Sánchez *et al.*'s approach (EFSM from statecharts and FTs) [127]. First, we compare the number of nodes of three case studies:

1. The railroad crossing control system presented here (RRCCS) in [6],
2. The gas burner example (GB) of [52], and
3. The launch vehicle (LV) in [54].

Table 4.1 shows this comparison. The left column identifies the case study. The column labeled BM reports the number of states (S) and transitions (T) in the behavioral model, respectively. The column labeled FM reports the number of leaves in the fault tree, and how many gates of various types are in the FT. The columns marked CEFSMs and EFSMs report the number of states and transitions in our approach vs. Sánchez *et al.*'s approach [127]. Note that our approach roughly increases states and transitions as a proportion of the number of leaves in the Fault Tree, while Sánchez *et al.*'s shows an exponential increase. Clearly, our approach looks more scalable. To investigate this further, we used our tool as a simulator with a range of model and fault tree sizes.

4.1.2.4 Simulation With Increasing Size

We fed the tool with input data of different size ranges of BM and FM. The behavioral models vary from 13 states and 15 transitions to 50 states and 60 transitions while the fault trees vary from 5 leaves to 19 leaves as shown in Table 4.2.

Table 4.1: Comparison

System	BM		FM					CEFSMs		EFSMs	
	S	T	leaves	AND	OR	XOR	Timing	S	T	Avg(S)	Avg(T)
GB	13	15	5	3	1	0	2	27	55	79	162
RRC	14	19	8	2	5	0	0	28	70	303	514
LV	21	39	14	0	10	0	0	41	117	4316	8335

S= State, T=Transition, Timing= Timing gates, Avg(S) = The average of the number of states and Avg(T) = The average of the number of transitions of 10 runs.

The fault tree is constructed of leaves which denote the number of input events to the fault tree and different types and numbers of gates, AND (0-6) gates, OR (1-10) gates, XOR (0-6) gates, and Timing gates (0-4) gates. The AND gate includes AND gate, Priority AND gate, and Inhibit gate.

We assume that the behavioral model is connected and no part of is isolated, therefore, the number of transitions must not be less than the number of states minus 1. We also assumed that the fault tree is a binary tree where the number of gates equals the number of leaf nodes minus 1. The timing gates, however, are excluded from this calculation because they take only one event and they appear only at the leaf nodes. However, we need to consider them to calculate the number of states and transitions of our integrated model, the ICEFSM. We assume that every behavioral model is integrated with every fault model. Therefore, the simulator calculates the size (states and transitions) of the integrated model of every behavioral model with every fault model as inputs.

We started with the relatively small behavioral model (GB) with 13 states and 15 transitions. This model is integrated with different fault trees as shown in Table 4.2. We can see that the number of states and transitions of the integrated model of the EFSM approach grows exponentially. The number of the states produced by

our integration approach grows from 21 to 59 and the number of transitions grows from 41 to 137, whereas the number of states produced by the EFSM integration approach grows on average from 79 to 70245 and transitions from 162 to 85330. It is very clear that the numbers of the states and transitions of both approaches are quite different.

Table 4.2: Simulation Data and Results

BM		FM					CEFSMs		EFSMs	
S	T	Leaves	AND	OR	XOR	Timing	S	T	Avg(S)	Avg(T)
13	15	5	3	1	0	2	27	55	79	162
13	15	7	3	2	1	1	28	64	178	304
13	15	8	2	5	0	0	27	66	263	416
13	15	14	0	10	0	0	33	93	2672	3280
13	15	19	6	5	6	4	59	158	18264	21342
13	17	5	3	1	0	2	27	57	79	174
13	17	7	3	2	1	1	28	66	178	332
13	17	8	2	5	0	0	27	68	263	456
13	17	14	0	10	0	0	33	95	2672	3691

Continued on next page

Continued from previous page

BM		FM					CEFSMs		EFSMs	
S	T	Leaves	AND	OR	XOR	Timing	S	T	Avg(S)	Avg(T)
13	17	19	6	5	6	4	59	160	18264	24152
15	19	5	3	1	0	2	29	59	92	197
15	19	7	3	2	1	1	30	68	206	374
15	19	8	2	5	0	0	29	70	303	514
15	19	14	0	10	0	0	35	97	3083	41335
15	19	19	6	5	6	4	61	162	21074	27003
17	18	5	3	1	0	2	31	58	104	202
17	18	7	3	2	1	1	32	67	233	376
17	18	8	2	5	0	0	31	69	343	5115
17	18	14	0	10	0	0	37	96	3494	3958
17	18	19	6	5	6	4	63	161	23883	25640
19	19	5	3	1	0	2	33	59	116	219

Continued on next page

Continued from previous page

BM		FM					CEFSMs		EFSMs	
S	T	Leaves	AND	OR	XOR	Timing	S	T	Avg(S)	Avg(T)
19	19	7	3	2	1	1	34	68	260	405
19	19	8	2	5	0	0	33	70	384	549
19	19	14	0	10	0	0	39	97	3905	4194
19	19	19	6	5	6	4	65	162	26693	27086
21	39	5	3	1	0	2	35	79	128	352
21	39	7	3	2	1	1	36	88	288	694
21	39	8	2	5	0	0	35	90	424	971
21	39	14	0	10	0	0	41	117	4316	8335
21	39	19	6	5	6	4	67	182	29503	55225
30	40	5	3	1	0	2	44	80	183	408
30	40	7	3	2	1	1	45	89	411	777
30	40	8	2	5	0	0	44	91	606	1069

Continued on next page

Continued from previous page

BM		FM					CEFSMs		EFSMs	
S	T	Leaves	AND	OR	XOR	Timing	S	T	Avg(S)	Avg(T)
30	40	14	0	10	0	0	50	118	6165	8677
30	40	19	6	5	6	4	76	183	42147	56817
40	45	5	3	1	0	2	54	85	244	493
40	45	7	3	2	1	1	55	94	548	921
40	45	8	2	5	0	0	54	96	808	1257
40	45	14	0	10	0	0	60	123	8220	9858
40	45	19	6	5	6	4	86	188	56196	64049
50	60	5	3	1	0	2	64	100	305	639
50	60	7	3	2	1	1	65	109	685	1204
50	60	8	2	5	0	0	64	111	1010	1648
50	60	14	0	10	0	0	70	138	10275	13093
50	60	19	6	5	6	4	96	203	70245	85330

S= State, T=Transition, Timing= Timing gates, Avg(S) = The average of the number of states and Avg(T) = The average of the number of transitions of 10 runs.

Table 4.2 shows that even for the larger BMs and larger Fault Trees with more

leaves, our approach produces integrated models of efficient sizes while the approach by Sánchez *et al.* very quickly reaches scalability limits. Figures 4.1-4.8 show the growth of the integrated models as a function of the number of leaves in the Fault Tree. While Sánchez *et al.*'s approach is highly affected by the number of leaves in the Fault Tree, our approach is not. As Figure 4.1, Figure 4.3, Figure 4.5, and 4.7 illustrate, CEFSM states and CEFSM transitions curves are invisible when we used the full simulation data because the numbers are so much smaller. Therefore, we represent these figures in Figure 4.2, Figure 4.4, Figure 4.6, and Figure 4.8 using only a part of the simulation data (no more than 8 leaf nodes for the failure model for every integrated model) in order to show the CEFSM states and CEFSM transitions for every simulation result.

Figure 4.8 represents the (50 state 60 transition) model. We can clearly notice that the trend of the curves in Figure 4.2 and Figure 4.8 are the same. The only difference is the number of states and transitions which depends on the size of the behavioral and the fault models.

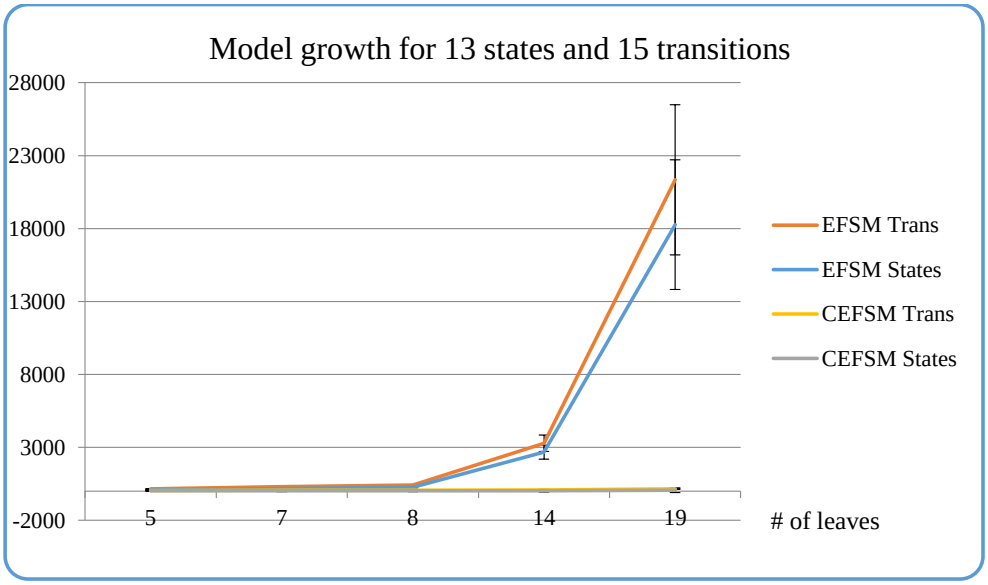

Figure 4.1: EFSM and CEFSM Approaches Model Growth for 13 S and 15 T BM (Full simulation data)

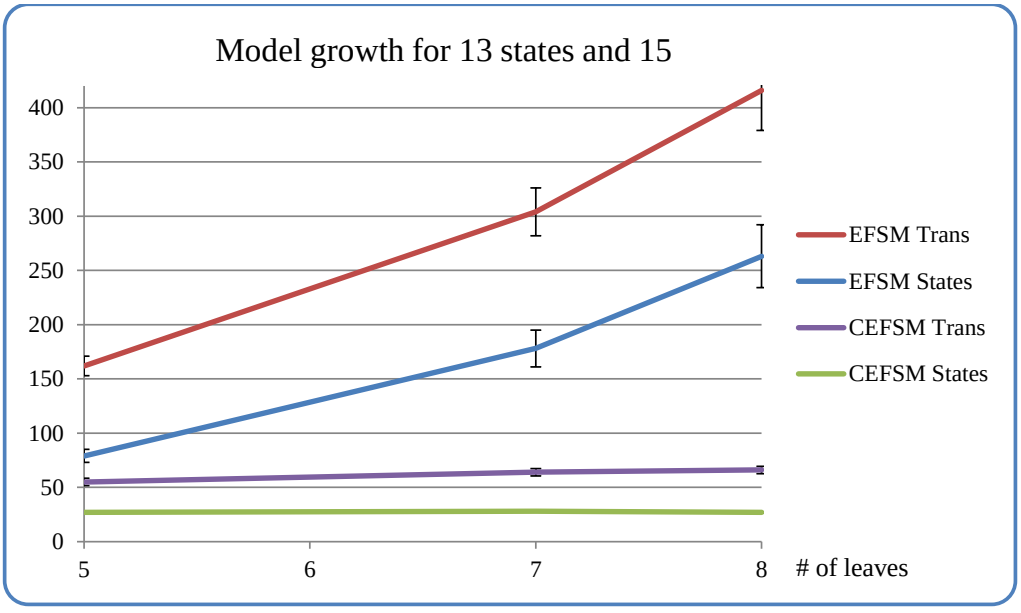

Figure 4.2: EFSM and CEFSM Approaches Model Growth for 13 S and 15 T Behavioral Model (up to 8 leaves)

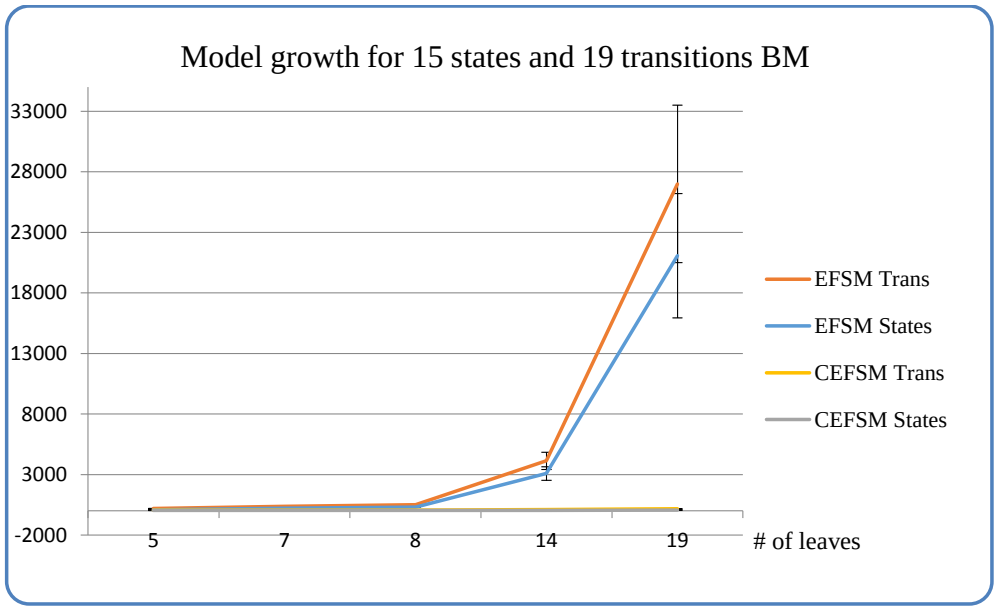

Figure 4.3: EFSM and CEFSM Approaches Model Growth for 15 S and 19 T BM (Full simulation data)

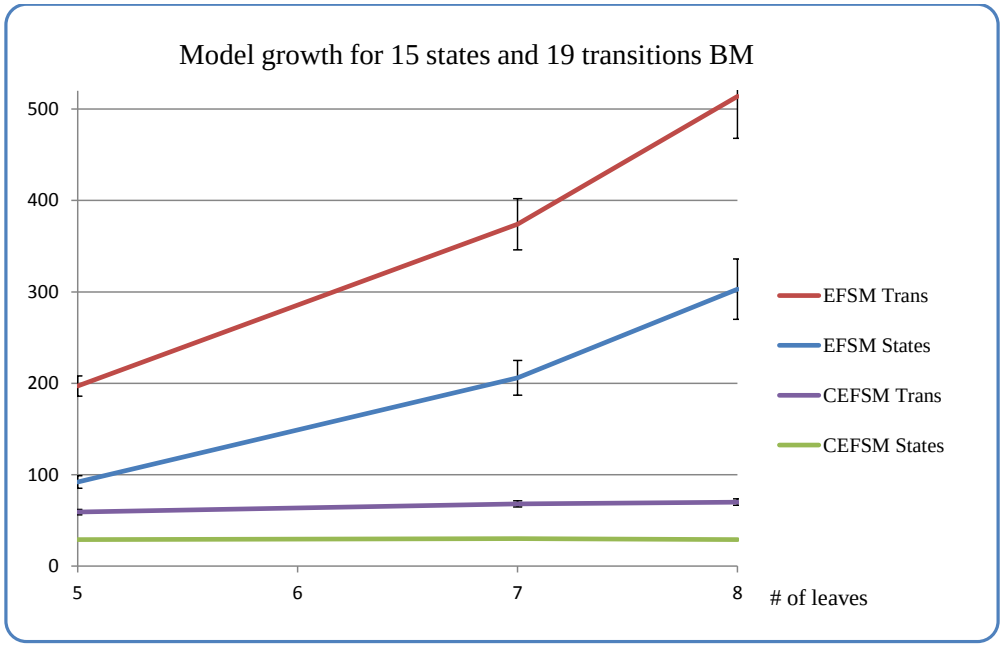

Figure 4.4: EFSM and CEFSM Approaches Model Growth for 15 S and 19 T Behavioral Model (up to 8 leaves)

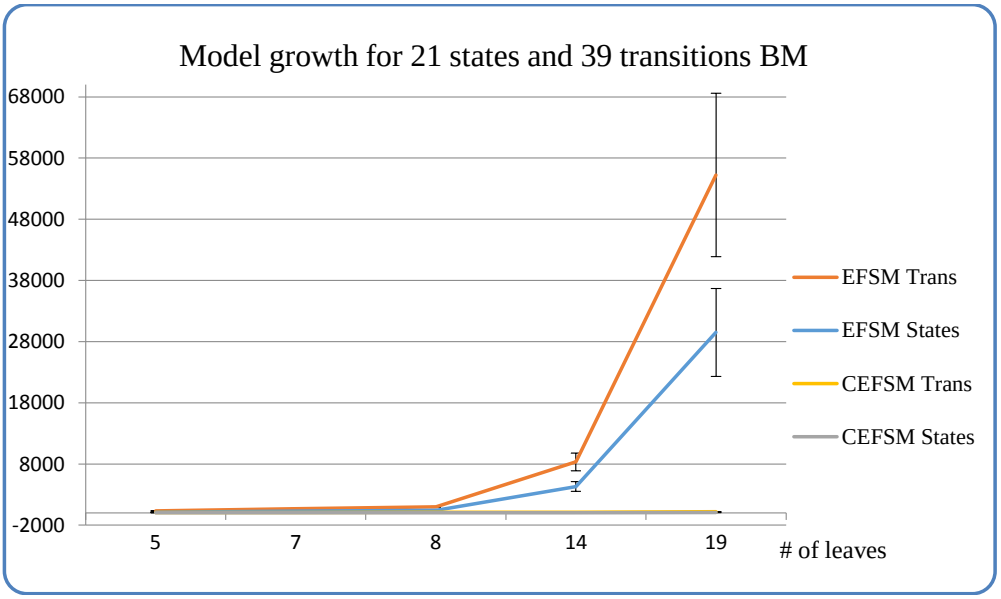

Figure 4.5: EFSM and CEFSM Approaches Model Growth for 21 S and 39 T BM (Full simulation data)

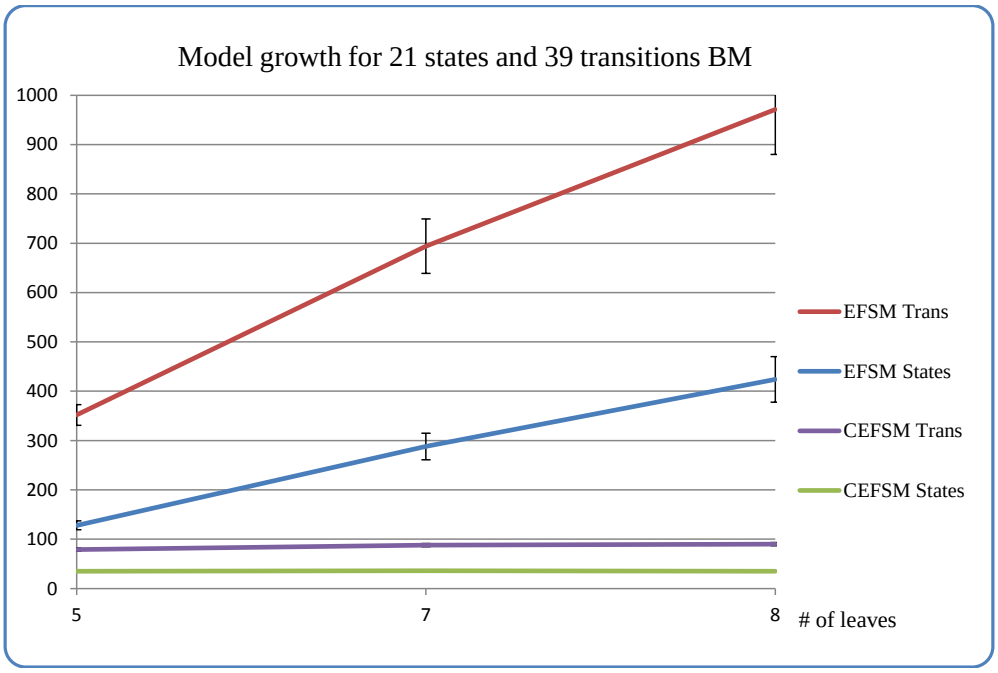

Figure 4.6: EFSM and CEFSM Approaches Model Growth for 21 S and 39 T Behavioral Model (up to 8 leaves)

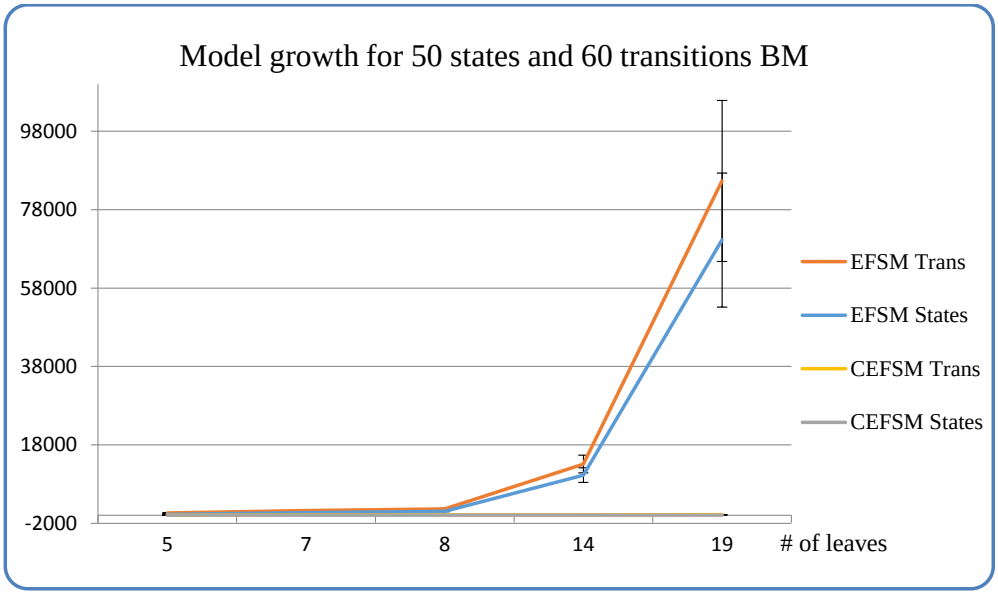

Figure 4.7: EFSM and CEFSM Approaches Model Growth for 50 S and 60 T BM (Full simulation data)

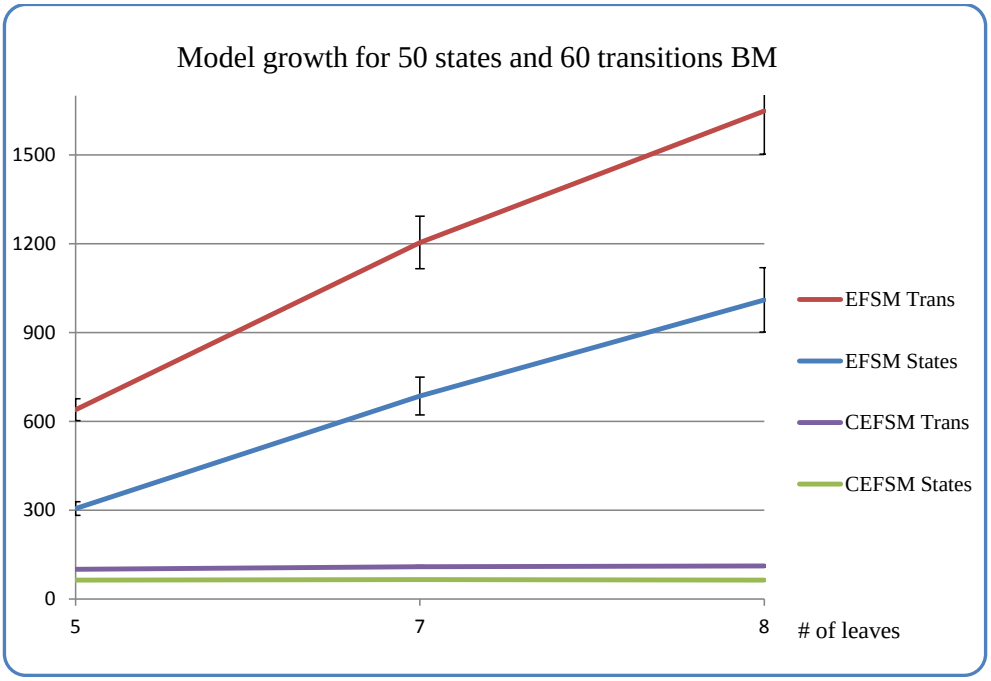

Figure 4.8: EFSM and CEFSM Approaches Model Growth for 50 S and 60 T Behavioral Model (up to 8 leaves)

4.2 Applicability: Case Studies

In this section, we apply our approach to three case studies of different sizes: a Gas Burner system [52], a Railroad Crossing Control System (RCCS) [53], and an Aerospace Launch System [54]. Seana Hagerman provided the functional description of the Launch system, the types of failures, and required mitigation. The variation of sizes, fault trees and mitigation requirements as well as using varying domains show the applicability of our approach. The Gas Burner System [52] is relatively a small system. It consists of five CEFSMs with a total of 13 states, 15 transitions, and 4 CEFSM communication channels. This behavioral model is integrated with a 4-gate-5-leaf fire occurrence fault tree. The RCCS [53] is another model from different application domain. It consists of four CEFSMs with a total of 14 states, 18 transitions, and 3 CEFSM communicating messages. This model is integrated with a 7-gate-8-leaf accident occurrence fault tree. The third model is the launch system. This system contains 5 CEFSMs with a total of 21 states, 34 transitions, and 5 CEFSM communication channels. This model is integrated with 4 fault trees altogether. These fault trees contain a total of 6 gates and 14 leaf nodes.

4.2.1 Gas Burner System

4.2.1.1 Description of Gas Burner System

We adapted the gas burner model of the example from [40] to explain how the transformed model will look like in CEFSM. Figure 4.9 shows the model of the gas burner system and Figure 4.10 shows the FT for the fire occurrence. The purpose of a gas burner is to produce heat by consuming gas. The model of the system consists of a controller component that controls the heat process and monitors a

gas valve (responsible for gas supply), an air valve (responsible for air supply), an igniter (responsible for the ignition), and flame detector (monitors the state of the flame) components.

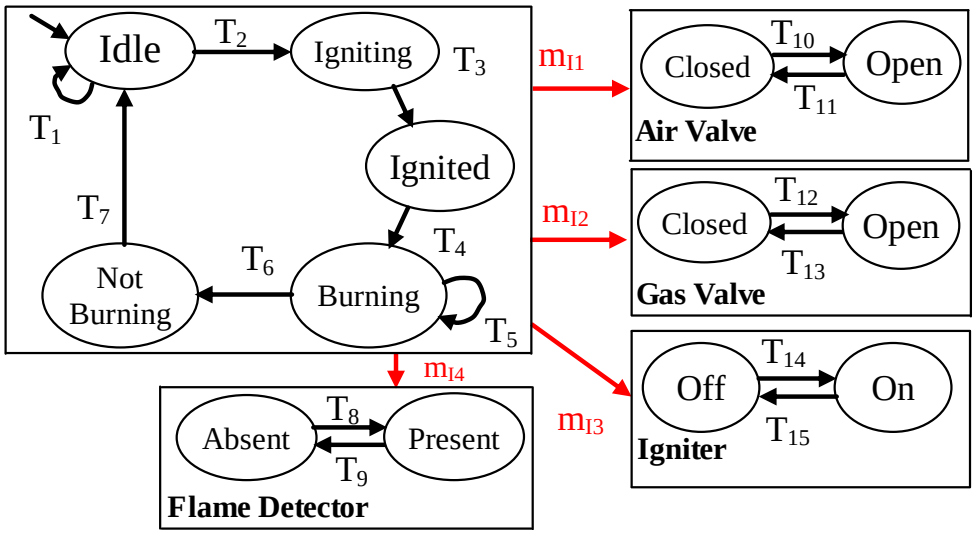

Figure 4.9: Gas Burner Model

Table 4.3: CEFSM model for a Gas Burner System Transitions

- **T_1**:(Idle,[NoheatReg=t],-)/(Idle,-,-)
- **T_2**:(Idle,HeatReg)/(Igniting,-,send(AirOn,GasOn,IgniteOn))
- **T_3**:(Igniting,-,-)/(Ignited,-,-)
- **T_4**:(Ignited,-,-)/(Burning,-,Send(IgniteOff))
- **T_5**:(Burning,[HeatReg=t&FlameOn=t])/(Burning,-,-)
- **T_6**:(Burning,[NoHeatReq=t|NoFlame=t],SetTimer(t,1))/(NotBurning,-,-)
- **T_7**:(NotBurning,TimeOut)/(Idle,-,Send(GasOff))
- **T_8**:(Absent,FlameOn)/(Present,-,Send(FlameOn))

Continued on next page

Continued from previous page
•$\mathbf{T_9}$:(Present,FlameOff)/(Absent,-,Send(FlameOff))
•$\mathbf{T_{10}}$:(Closed,AirOn)/(Open,-,Send(m_{f1}"AirOn"))
•$\mathbf{T_{11}}$:(Open,AirOff)/(Closed,-,Send(m_{f1}"AirOff"))
•$\mathbf{T_{12}}$:(Closed,GasOn)/(Open,-,Send(m_{f2}"GasOn",m_{f3}"GasOn"))
•$\mathbf{T_{13}}$:(Open,GasOff)/(Closed,-,Send(m_{f2}"GasOff",m_{f3}"GasOff"))
•$\mathbf{T_{14}}$:(Off,IgniteOn)/(On,-,Send(m_{f4}"IgniteOn"))
•$\mathbf{T_{15}}$:(On,IgniteOff)/(Off,-,Send(m_{f4}"IgniteOff"))

4.2.1.2 Gas Burner Failure

The fault model shown in Figure 4.10 describes fire occurrence and the events that contribute to fire occurrence when they occur. If the gas leaks for more than 4 seconds during an interval window of less than 30 seconds, it means that there is an excess of gas which, if combined with the presence of gas, causes an unsafe environment. If an ignition is attempted when there is an unsafe environment, a fire will occur.

4.2.1.3 Compatibility Transformation Step

The first step is the compatibility transformation. At this step we create *Bclass* and *Fclass* for the failure related entities *GasValve*, *AirValve*, and *Igniter* and combine the related classes according to the compatibility transformation procedure 3.1.3. These classes are shown in Figure 4.11, Figure 4.12, and Figure 4.13. Table 4.4 shows the composition of the FT′ from these classes at each gate. The events in the FT are substituted with the combined attributes from the *BF classes* that are

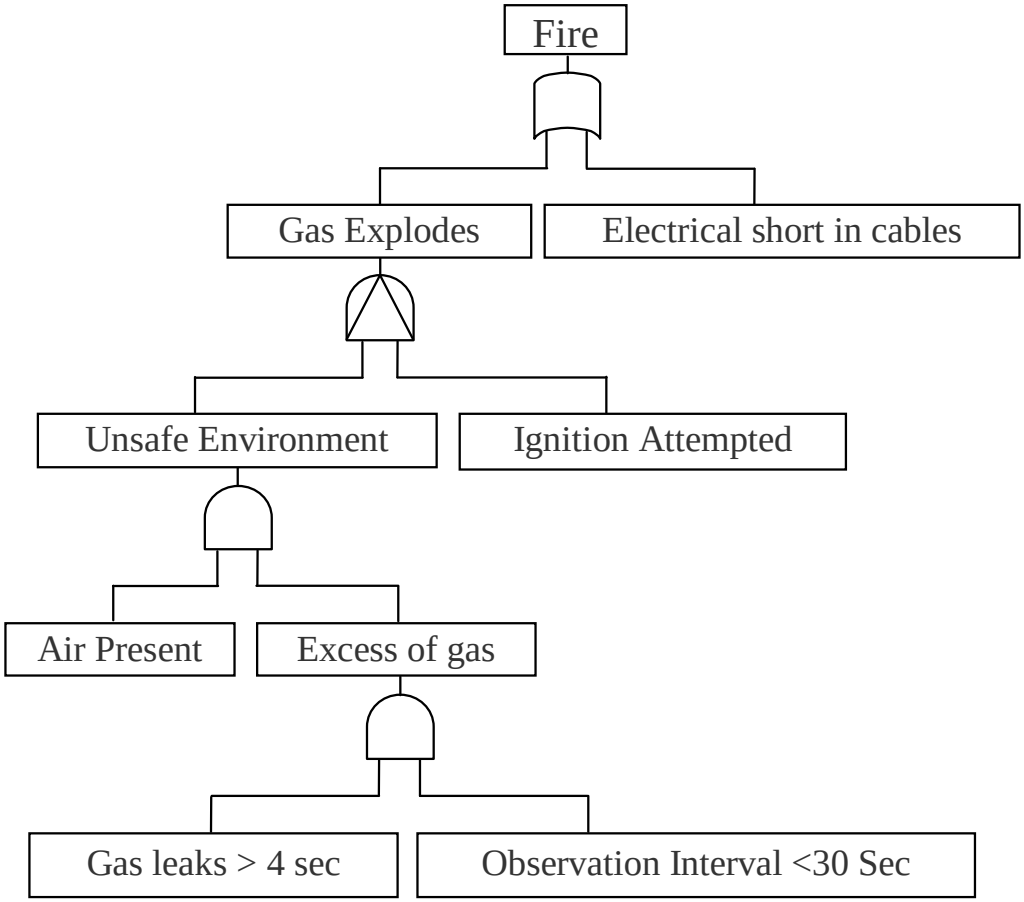

Figure 4.10: FT for a Fire Occurrence in a Gas Burner [40]

equivalent to these events. For example, the event *Air present* in the FT is equivalent to *BFAirValve.FeventCond* in FT′. The attributes of *BAirValve* and *FAirValve* are combined in *BFAirValve*. As we can see in Fig 4.11, the attribute *BState* belongs to the class *BAirValve* at the behavioral model and *FState* belongs to the *FAirValve* at the fault model. For example, the event *Gas leaks > 4 sec* in the FT is equivalent to *BFGasValve.FeventCond* and the event *Observation Interval < 30 sec* is equivalent to *BFObservation.FEventCond*.

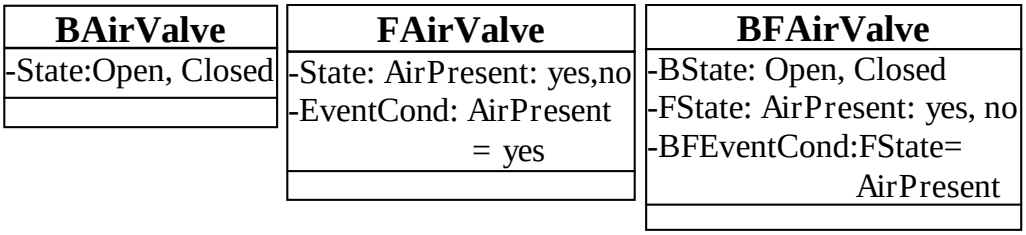

Figure 4.11: Bclass, Fclass, and BFclass for AirValve Entity

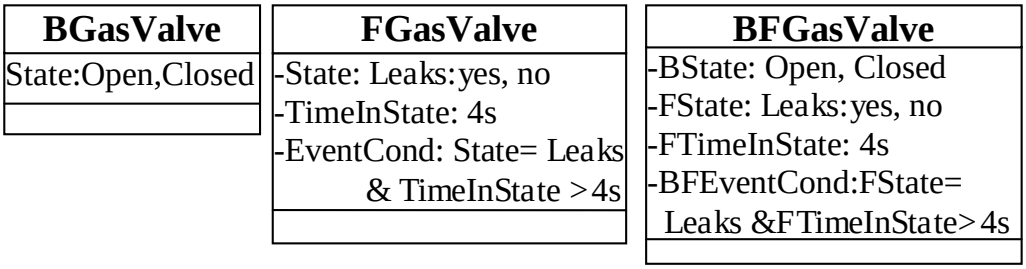

Figure 4.12: Bclass, Fclass, and BFclass for GasValve Entity

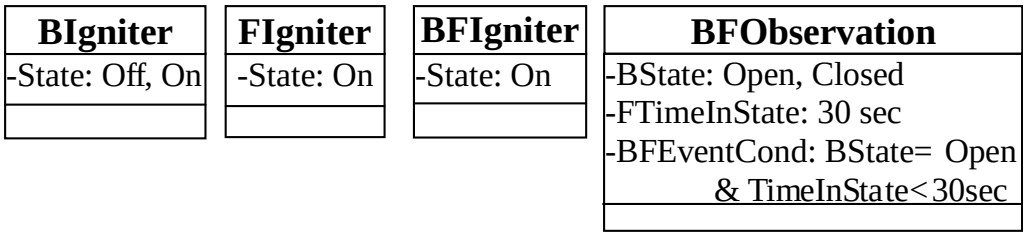

Figure 4.13: Igniter and Observation Classes

After the compatibility transformation procedure is finished, the complete FT′ will be: $(\vee,(\overline{\wedge},(\wedge,$*BFAirValve.FEventCond*, $(\wedge,$*BFGasValve.FeventCond*, *BFObservation.FeventCond*$)),$*BFIgniter.State*$),$*ElectricalShortInCable*$)$.

4.2.1.4 Fault Tree Transformation

The fault CEFSM is constructed according to a tree postorder traversal. The FT is read gate by gate starting from the root node until we reach the leftmost leaf

Table 4.4: BFClass

Name	Formula	GCEFSM
Excess of gas	$=BFGasValve.FeventCond \wedge BFObservation.\ FeventCond$	Figure 4.16
Unsafe Environment	$=BFAirValve.FEventCond \wedge$ *Excess of gas*	Figure 4.17
Gas Explodes	$=$*Unsafe Environment* $\bar{\wedge}$ *BFIgniter.State*	Figure 4.18
Fire	$=$*Gas Explodes* $\vee$ *Electrical short in cable*	Figure 4.19

node. The transformation starts with the leftmost leaf of the FT which is in this example "air present". The event is described in terms of class diagram as shown in Figure 4.11.

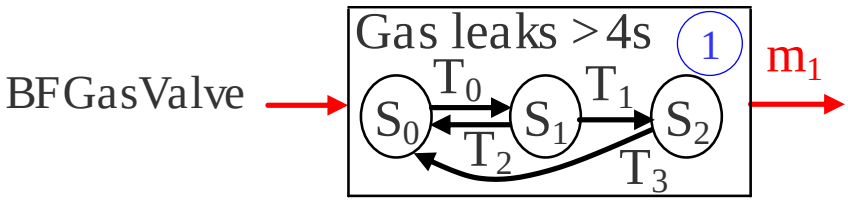

Figure 4.14: Event Timing GCEFSM for Gas Leaks > 4s

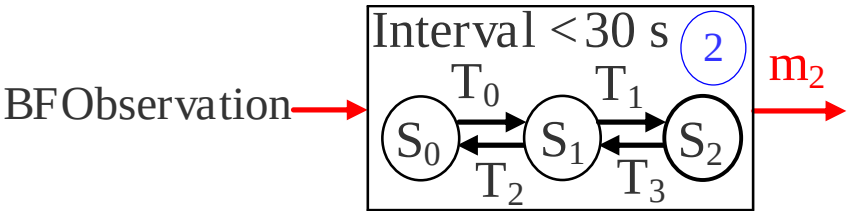

Figure 4.15: GCEFSM for Gas Observation Interval < 30s

Next we look for the right sibling of this gate which turns to be an AND gate between two events. The left child of this node is an event but it is not simple. It is composed of a timed event. In this case we need to use the "event timer" gate we presented in the transformation rules after configuring the value of the timer

and the outgoing message number. The message *id* should carry the same number as the gate. In this case the gate is given number one since it is the first gate to transform. The numbering of the internal transition is not important since each gate is an independent entity and no confusion will occur. The gate is shown in Figure 4.14. The right child also is a composed event. It is an event timed for continuous time intervals, in which we use the timing continuous interval gate, give it a number (number 2 since it is the second gate transformed), and create the input and output messages. Table 4.5 shows the event-gate table at this point. The gate is shown in Figure 4.15.

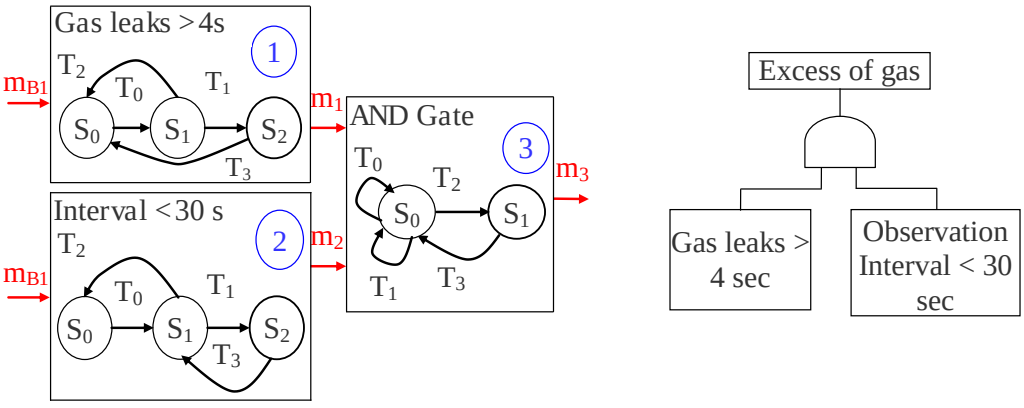

Figure 4.16: GCEFSMs for Excess Of Gas

Table 4.5: Event-Gate Table

Event name & attribute	Event ID	Gate ID
BFGasValve.BFEventCond	e_{B1}	G_1
BFObservation.BFEventCond	e_{B2}	G_2
BFAirValve.FEventCond	e_{B3}	G_4
BFIgniter.State	e_{B4}	G_5

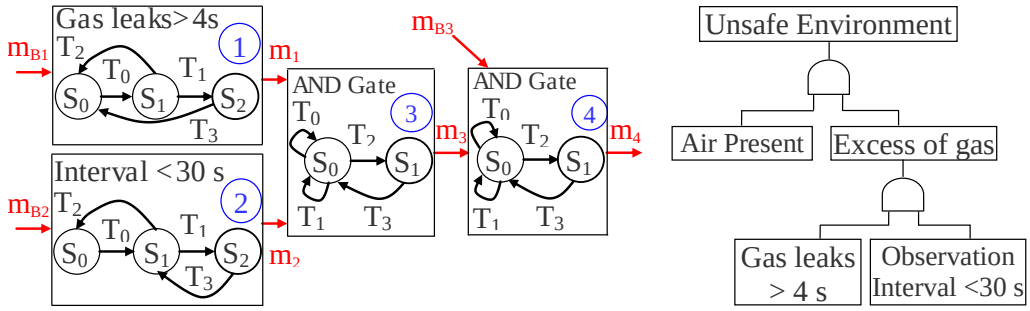

Figure 4.17: GCEFSMs for Unsafe Environment

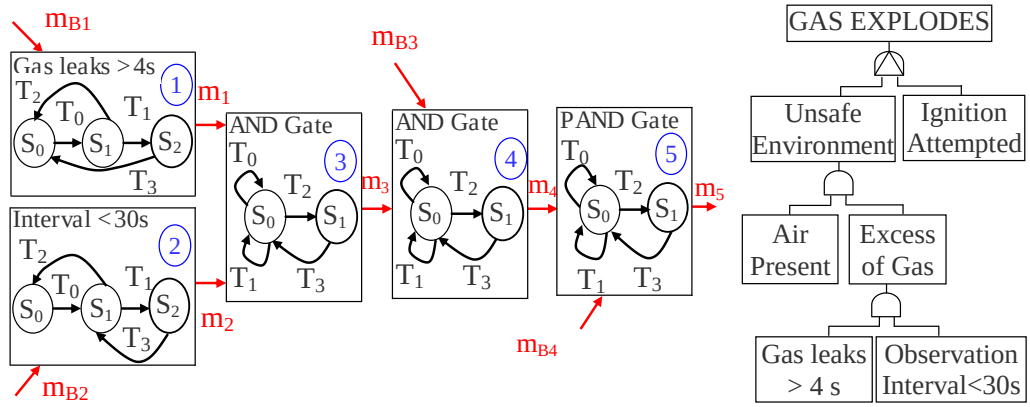

Figure 4.18: GCEFSMs for Gas Explodes

At this point the Event-Gate table contains the entries of the leaf nodes that were found so far as shown in Table 4.5. Since there are no other children for this AND gate, we transform the gate itself. We use the predefined representation for AND gate from the transformation rules. Figure 4.16 shows the part of the fault tree that has been transformed, the AND gate and its inputs and output. The transformed AND gate is a right child of another AND gate, that is the gate between "Air present" and "Excess of gas" events. "Air present" is a simple event from the behavioral model while the "Excess of gas" event, which is represented as "m_3", is the output message of this AND gate. The next step is to transform the AND gate that combines "Air present AND Excess of gas". The same transformation steps are followed and this gate is given number 4. The inputs of this gate are m_{B3} and

m_3 messages which are equivalent to "Air present" and "Excess of gas" respectively. Figure 4.17 shows the transformed gates. The next gate to be transformed is the Priority AND gate which combines the "Unsafe Environment" and "Ignition Attempted" events. For this gate the order in which these events occur is important and defined as left to right order.

In this FT example, The left event, "Unsafe Environment", should occur before the event "Ignition Attempted". Therefore, this order is considered in the GCEFSM PAND gate. Figure 4.18 illustrates the GCEFSM after the PAND gate is transformed. The event m_{B4} represents the event "Ignition Attempted" in the FT, which is a message received directly from the behavioral model by this gate indicating that the igniter is on or off. This algorithm continues until the whole FT is transformed.

4.2.1.5 Model Integration

After the fault tree is transformed to GCEFSMs, we start integrating it into the behavioral model. At this point, every message in the BM contains an event name that is related to an event in one of the leaf nods of the fault tree. We check the class diagram and the Event-Gate table to find the event ID and the gate ID for the event. These event ID and gate ID are inserted into the message at the BM. The event *Gas leak > 4 sec* is represented in the class diagram as *BFGasValve.FeventCond*. This event is looked up inside the the event-gate table to get its event ID (e_{B1}) and the gate ID (G_1) the message is sent to. The message in the BM is modified as (m_{B1}, e_{B1}, G_1). Then, the event *Observation Interval < 30 Sec* which is represented in the class diagram as *BFObservation.BFEventCond* is looked up inside the the event-gate table to get its event ID (e_{B2}) and the gate ID (G_2) the message is sent to. The message in the BM is modified as (m_{B2}, e_{B2}, G_2). Next, the event *Air*

Present which is represented in the class diagram as *BFAirValve.FEventCond* is looked up to get its event ID (e_{B3}) and the gate ID (G_4) the message is sent to. The message in the BM is modified as (m_{B3}, e_{B3}, G_4). Finally, the event *Ignition Attempted* which is represented in the class diagram as *BFIgniter.State* is looked up to get its event ID (e_{B4}) and the gate ID (G_5) the message is sent to. The message in the BM is modified as (m_{B4}, e_{B4}, G_5).

Figure 4.19 illustrates the gas burner system transformed to an CEFSM model integrated with a transformed FT (GCEFSMs). There are two connected models, the behavioral model and the FT model. The red arrows represent the communicating messages between the CEFSMs. The transformed system shown in Figure 4.19 forms a graph to which suitable coverage criteria can be applied. The FT gates that are directly connected to the behavioral model receive messages from the behavioral model and acts accordingly. The messages m_1 to m_5 represent the global transitions between the GCEFSMs for the FT part, while m_{I1} to m_{I4} represent the messages between the components of the behavioral model and m_{B1} to m_{B5} represent the communicating messages between the BM and FT. If we apply the algorithm in [68] on the graph in Figure 4.19 by imposing the edge coverage criteria on the global transitions of the ICEFSM, we will get the test paths shown in Table 4.7.

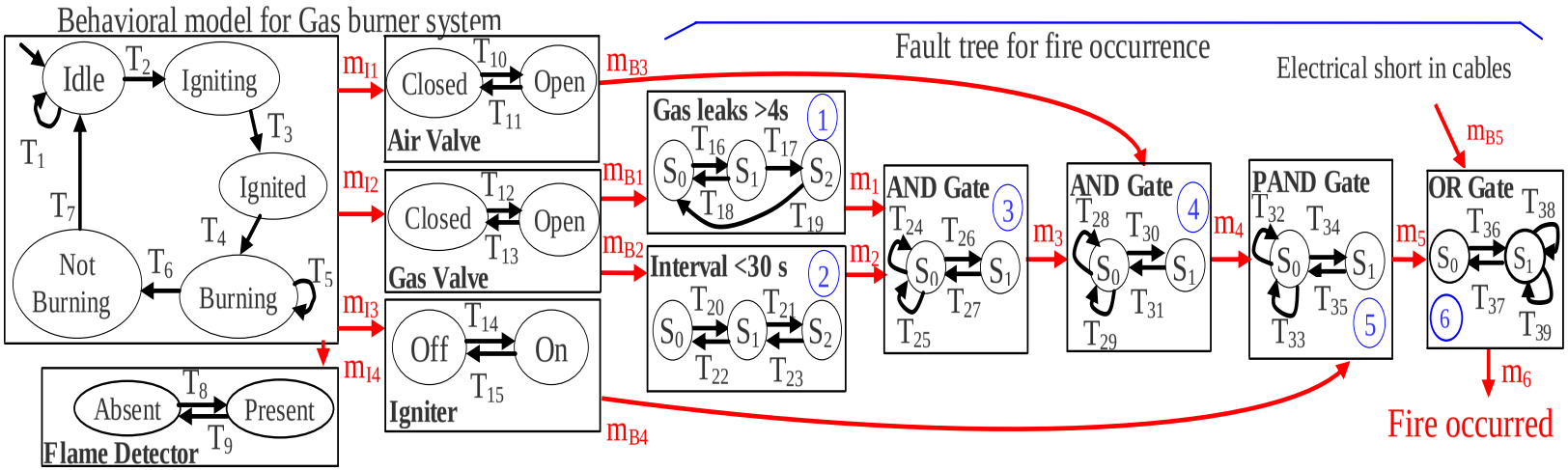

Figure 4.19: ICEFSM Model for a Gas Burner System

Table 4.6: ICEFSM model for a Gas Burner System Transitions

•$\mathbf{T_1}$:(Idle,[NoheatReg=*true*],-)/(Idle,-,-)
•$\mathbf{T_2}$:(Idle,HeatReg)/(Igniting, -, send(AirOn,GasOn,IgniteOn))
•$\mathbf{T_3}$:(Igniting,-,-)/(Ignited,-,-)
•$\mathbf{T_4}$:(Ignited,-,-)/(Burning, -, Send(IgniteOff))
•$\mathbf{T_5}$:(Burning,[HeatReg =*true*&FlameOn = *true*])/(Burning, -, -)
•$\mathbf{T_6}$:(Burning,[NoHeatReq =*true*|NoFlame = *true*],SetTimer(t,1))/ (NotBurning,-,-)
•$\mathbf{T_7}$:(NotBurning, TimeOut)/(Idle, -, Send(GasOff))
•$\mathbf{T_8}$:(Absent, FlameOn)/(Present, -, Send(FlameOn))
•$\mathbf{T_9}$:(Present, FlameOff)/(Absent, -, Send(FlameOff))
•$\mathbf{T_{10}}$:(Closed, AirOn)/(Open, -, Send(m_{B3}))
•$\mathbf{T_{11}}$:(Open, AirOff)/(Closed, -, Send(m_{B3}))
•$\mathbf{T_{12}}$:(Closed, GasOn)/(Open, -, Send(m_{B1},m_{B2}))
•$\mathbf{T_{13}}$:(Open, GasOff)/(Closed, -, Send(m_{B1},m_{B2}))
•$\mathbf{T_{14}}$:(Off, IgniteOn)/(On, -, Send(m_{B4}))
Continued on next page

Continued from previous page

- **T_{15}**:(On, IgniteOff)/(Off, -, Send(m_{B4}))
- **T_{16}**:(S_0,setTimer(v,4s))/(S_1, -, -)
- **T_{17}**:(S_1,timeout)/(S_2, -, Send(GasleakingMSG))
- **T_{18}**:(S_1, [e_i.eId = "GasNotLeaking"], get(m_{B1}))/ (S_0, reset(Timer), Send(GateNotOccured))
- **T_{19}**:(S_2, e_i.eStatus = *false*], get(m_{B1}))/ (S_0, reset(Timer); update(events),Send(GateNotOccurred))
- **T_{20}**:(S_0, get(m_{B2})/(S_1, setTimer(v,30), Send(GateOccurred))
- **T_{21}**:(S_1, timeout)/(S_2, -, Send(GateNotOccurred))
- **T_{22}**:(S_1,[e_i.eId = "GasOff"], get(m_{B2}))/(S_0, reset(v,30s), Send(GateNotOccurred))
- **T_{23}**:(S_2, setTimer(v,30s))/(S_1, -, Send(GateOccurred))
- **T_{24}**:(S_0,[NoOfPositiveEvents $<$ 2 & e_i.eStatus = *true*], get(m_i))/(S_0, update(events),-)
- **T_{25}**:(S_0,[NoOfPositiveEvents $>$ 0 & e_i.eStatus = *false*], get(m_i))/(S_0, update(events),-)
- **T_{26}**:(S_0,[NoOfPositiveEvents = 2 & e_i.eStatus = *true*],get(m_i))/(S_1, update(events),Send(GateOccurred))

Continued on next page

Continued from previous page

- **T_{27}**:(S_1,[e_i.eStatus = t], get(m_i))/(S_0,update(events),Send(Gate NotOccurred))

- **T_{28}**:(S_0,[NoOfPositiveEvents < 2 & e_i.eStatus = *true*],get(m_i))/(S_0, update(events),-)

- **T_{29}**:(S_0,[NoOf PositiveEvents > 0 & e_i.eStatus = *false*], get(m_i))/(S_0, update(events),-)

- **T_{30}**:(S_0,[NoOfPositiveEvents = 2 & e_i.eStatus = *true*],get(m_i))/(S_1, update(events),Send(GateOccurred))

- **T_{31}**:(S_1,[e_i.eStatus = *false*], get(m_i))/(S_0,update(events),Send(GateNot Occurred))

- **T_{32}**:(S_0,[ordered(input);notLast(input)], get(m_i))/(S_0,update(event),-)

- **T_{33}**:(S_0, [e_i.eId = "GateNotOccurred"|e_i.eId = "igniteOff"],get(m_i))/ (S_0,update(event),-)

- **T_{34}**:(S_0,last(input), get(m_i))/(S_1, update(events), Send(GateOccurred))

- **T_{35}**:(S_1,[e_i.eId = "GateNotoccurred"|e_i.eId="igniteOff"],get(input))/ (S_0, update(events), Send(GateNotOccurred))

- **T_{36}**:(S_0,[e_i.eId = "GateOccurred"|e_i.eId="ElectricalShortInCables"], get(m_i))/(S_1,-,Send(GateOccurred))

- **T_{37}**:(S_1,[e_i.eStatus = *false* & NoOfPositiveEvents = 0],get(m_i))/(S_1,-, Send(GateNot Occurred))

Continued on next page

Continued from previous page

- **T_{38}**:(S_1, [NoOfPositiveEvents = 2 & e_i.eStatus = *true*],get(m_i))/(S_1, update(events),-)
- **T_{39}**:(S_1, [e_i.eStatus = *false*], get(m_i))/(S_1,update(events),-)
- m_{B1}:(m_{B1},e_1, G_1)
- m_{B2}:(m_{B2},e_2, G_2)
- m_{B3}:(m_{B3},e_{B3}, G_4)
- m_{B4}:(m_{B4},e_{B4}, G_5)

Table 4.7: Gas Burner System Test Paths

- Idle$\xrightarrow{T_1}$Idle
- Idle$\xrightarrow{T_2}$Igniting$\xrightarrow{T_3}$Ignited$\xrightarrow{T_4}$Burning$\xrightarrow{T_5}$Burning$\xrightarrow{T_6}$NotBurning$\xrightarrow{T_7}$Idle
- Idle$\xrightarrow{m_{I3}}$Igniter[Off$\xrightarrow{T_{14}}$On]$\xrightarrow{m_{B4}}$5$\xrightarrow{m_5}$6
- Idle$\xrightarrow{T_2}$Igniting$\xrightarrow{T_3}$Ignited$\xrightarrow{m_{I3}}$Igniter[On$\xrightarrow{T_{15}}$Off]$\xrightarrow{m_{B4}}$5$\xrightarrow{m_5}$6
- Idle$\xrightarrow{m_{I1}}$AirValve[Closed$\xrightarrow{T_{10}}$Open]$\xrightarrow{m_{B3}}$4$\xrightarrow{m_4}$5$\xrightarrow{m_5}$6
- Idle$\xrightarrow{m_{I2}}$GasValve[Closed$\xrightarrow{T_{12}}$Open]$\xrightarrow{m_{B2}}$2$\xrightarrow{m_2}$3$\xrightarrow{m_3}$4$\xrightarrow{m_4}$5$\xrightarrow{m_5}$6
- Idle$\xrightarrow{m_{I2}}$GasValve[closed$\xrightarrow{T_{12}}$Open]$\xrightarrow{m_{B1}}$1$\xrightarrow{m_1}$3$\xrightarrow{m_3}$4$\xrightarrow{m_4}$5$\xrightarrow{m_5}$6
- Idle$\xrightarrow{T_2}$Igniting$\xrightarrow{T_3}$Ignited$\xrightarrow{T_4}$Burning$\xrightarrow{T_5}$Burning$\xrightarrow{T_6}$NotBurning $\xrightarrow{m_{I2}}$GasValve[Open$\xrightarrow{T_{13}}$Closed]$\xrightarrow{m_{B1}}$1$\xrightarrow{m_1}$3$\xrightarrow{m_3}$4$\xrightarrow{m_4}$5$\xrightarrow{m_5}$6
- Idle$\xrightarrow{T_2}$Igniting$\xrightarrow{T_3}$Ignited$\xrightarrow{T_4}$Burning$\xrightarrow{T_5}$Burning$\xrightarrow{T_6}$NotBurning$\xrightarrow{m_{I2}}$ GasValve[Open$\xrightarrow{T_{13}}$Closed]$\xrightarrow{m_{B2}}$2$\xrightarrow{m_2}$3$\xrightarrow{m_3}$4$\xrightarrow{m_4}$5$\xrightarrow{m_5}$6

The difference between our approach and those that use statecharts such as [127, 40, 82] is that our approach is used to explicitly model systems (with communication edges) where the behavior process and the failure process intersect. Therefore, paths can be produced. It is also possible to manipulate sensor values and create

events during system testing. This model can also be used as a simulation test bed. Moreover, in our approach, different levels of details can be used for different testing purposes. For example, if we want to test the system, we can look at every GCEFSM as a whole and we do not have to worry about the GCEFSMs' internal details (transitions and states) since we know how they behave. When we compared the number of states and transitions produced by our integration approach with those of [127] on this Gas burner example, we found that the ICEFSM contains 27 states and 41 transitions whereas the EFSM model of [127] will contain at least 84 states and 168 transitions. The slicing algorithm used in [127] will not be useful in partitioning the model here because the FT has two minimum cuts one of which contains all the leaf nodes except for the external event "Electrical short in cables".

4.2.2 Application: Aerospace Launch System

4.2.2.1 Description of Launch System

In this section we demonstrate our approach with a launch system example to show the integration of multiple fault trees [1] into CEFSMs. A launch system consists of a launch conductor, ground system, launch pad, mobile launch platform and a launch vehicle which is comprised of a booster, upper stage and a payload. The booster and upper stage are fueled by cryogenic fuels which can only be liquefied at extremely low temperatures. Cryogenic fuels are chosen because they generate a high specific impulse, which defines their efficiency of fuel relative to the amount consumed. A medium lift vehicle is capable of lofting a payload weighing between 4000 and 40,000 lbs. into low earth orbit. The launch controller is responsible for initiating the launch sequence and verifying the safety and security of the launch

[1] Seana Hagerman contributed to this case study

control system throughout the launch. The launch conductor communicates to the vehicle through the ground system. The ground system is physically connected to the launch vehicle via Ethernet cables, serial cables, 1553 data cables and fuel lines.

The sequence begins about 24 hours before a launch when the launch conductor initiates the countdown clock. The launch conductor then clears the area of non-essential personnel using a public announcement system. The mobile launch pad is prepared for jacking. The launch conductor initiates environmental control system (ECS) on the launch pad, solicits a weather briefing, and turns on both search lights and amber warning lights. The MLP and vehicle are moved to the launch pad. Cryogenic tanking begins on the launch vehicle and an instrumentation check is performed. A test to detect hazard gas is performed. The launch vehicle's Liquid Oxygen LO2 is verified as well as the upper stage's Liquid Hydrogen LH2. The launch conductor periodically conducts polls of the stakeholders to obtain concurrence to continue the sequence. When concurrence is received, the launch conductor initiates the chill down procedures and flight pressures. The safe arm device (SAD) is initiated. The SAD is used to terminate the flight, should there be a problem after launch. The launch conductor commands the launch vehicle to switch to internal power and the vehicle lifts off the launch pad. Figure 4.20 shows the CEFSM model of the launch system including transitions, variables, events, and messages.

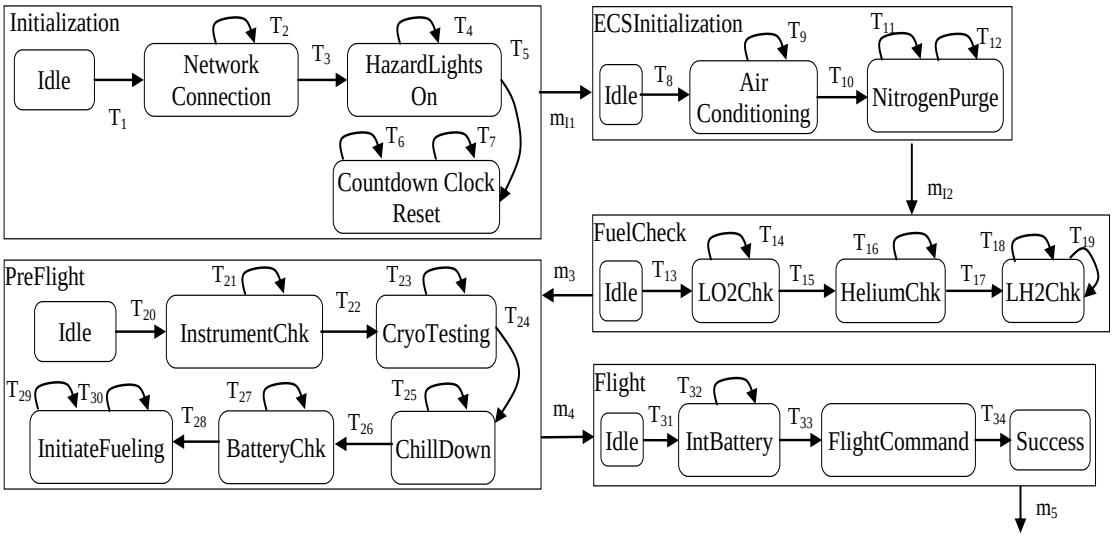

Figure 4.20: CEFSM Model for a Launch System

Table 4.8: CEFSM Model for a Launch System Transitions

- $\mathbf{T}_1$**:**(Idle,[startSequence=True],startConnection)/(NetworkConnection,-)
- $\mathbf{T}_2$**:**(NetworkConnection,[ConnectionConfirmed=false|timeout>= 30000])/(NetworkConnection,-,send(mf_2"NetworkConnectionfail"))
- $\mathbf{T}_3$**:**(NetworkConnection,[ConnectionConfirmed=True],TurnLightsOn)/ (HazardLightsOn,-,-)
- $\mathbf{T}_4$**:**(HazardsightsOn,[AllHazardLighsOn=false],)/(HazardLightsOn,-, send(mf_4"HazardLightsfail"))
- $\mathbf{T}_5$**:**(HazardLightsOn,[AllHazardLighsOn=true],ResetClock)/(Count DownClockReset,-,-)
- $\mathbf{T}_6$**:**(CountDownclockRwset,[ClkError=true],)/(CountDownclockReset , -,send(m_{I1}"startAC"))
- $\mathbf{T}_7$**:**(CountDownclocLReset,[ClkError=false],)/(CountDownclockReset ,-,send(mf_7"CLKFail"))

Continued on next page

Continued from previous page

- **T_8:**(Idle,get(startAC))/(AirConditioning,-,-)
- **T_9:**(AirConditioning,[ACError=true],purge)/(AirConditioning,-,send (mf_9"ACError"))
- **T_{10}:**(AirConditioning,[ACError=false],purge)/(NitrogenPurge,-,-)
- **T_{11}:**(NitrogenPurge, [ECSError = false])/(NitrogenPurge,-,send(mf_{11} "fuelcheckFail"))
- **T_{12}:**(NitrogenPurge, [ECSError = ture])/ (NitrogenPurge, -, send(mf_{I2} "LH 2Chk"))
- **T_{13}:**(Idle,,get(m_2"fuelcheck")/(LO2Chk,-,-)
- **T_{14}:**(LO2Chk,[LO2leak=true|LO2PressureOk=flase])/(LO2Chk,-,send(mf_{14}"LO2fail"))
- **T_{15}:**(LO2Chk,[LO2leak=false&LO2PressureOK=true])/(HeliumChk,-,-)
- **T_{16}:**(HeliumChk,[Heliumleak=true|HeliumPressureOK=false])/(Helium Chk,-,send(mf_{16}"Heliumfail"))
- **T_{17}:**(HeliumChk,[Heliumleak=false&HeliumPressureOK=true])/ (LH2Chk,-,-)
- **T_{18}:**(LH2Chk,[LH2leak=true|LH2PressureOk=false])/(LH2Chk,- send(mf18"Heliumfail"))
- **T_{19}:**(LH2Chk,[LH2leak=false&LH2PressureOK=true])/(LH2Chk,- send(m_{I3} "PreFlight"))
- **T_{20}:**(Idle,,get(m3"PreFlight")/(INSTChk,-,-)
- **T_{21}:**(INSTChk,[ChkcksumOK=false|LaunchConductCommOk =false])/ (INSTChk,-,send(mf_{21}"Instrufail"))

Continued on next page

Continued from previous page

- $\mathbf{T_{22}}$**:**(INSTChk,[ChecksumOk=true&LaunchConductCommOk=true])/ (CryoTesting,-,-)
- $\mathbf{T_{23}}$**:**(CryoTesting,[IntTempOK=false|IntPressureOk=false])/ (CryoTesting,-,send(mf_{23}"INSTfail"))
- $\mathbf{T_{24}}$**:**(CryoTesting,[IntTempOK=true&IntPressureOk=true]) / (ChillDown,-,-)
- $\mathbf{T_{25}}$**:**(ChillDown,[IntTempOK=false|InterPssurOK=false])/ (ChillDown ,-,send(mf_{25}"ChillDownfail"))
- $\mathbf{T_{26}}$**:**(ChillDown,[IntTemIOK=true&IntPressurOK=true])/ (BatteryChk,-,-)
- $\mathbf{T_{27}}$**:**(BatteryChk,[BatteryPresent = false|PowerLevelOK = false | BatteryLifeOK = false])/(BatteryChk,-,send(mf27"Batteryfail"))
- $\mathbf{T_{28}}$**:**(BatteryChk,[BatteryPresent = true&PowerLevelOK = true & BatteryLifeOK = true])/(InitiatFueling,-,-)
- $\mathbf{T_{29}}$**:**(InitiateFueling,[TankPressureOK=false|FuelLevelOK=false | TankTempOK = false])/(InitiatFueling,-,send(mf_{29}"Fuelingfail"))
- $\mathbf{T_{30}}$**:**(InitiateFueling,[TankPressureOK = true&FuelLevelOK = true & TankTempOK = true])/(InitiateFueling,-,send(m_{I4}"Flight"))
- $\mathbf{T_{31}}$**:**(Idle,,get(m_4" Flight")/(InternalBattery,-,-)
- $\mathbf{T_{32}}$**:**(InternalBattery,[SwitchToBatteryOK=false|PowerLevelOK=false])/ (InternalBattery,-,send(mf_{32}"InternalBatteryfail"))
- $\mathbf{T_{33}}$**:**(InternalBattery,[SwitchToBattOK=true&PowerLevelOK=true])/ (FlightCommand,-,-)
- $\mathbf{T_{34}}$**:**(FlightCommand,[StartFlight=true],StartFlight)/(Success,-,send(m_5))

4.2.2.2 Launch System Failure

The Aerospace launch system fault trees include initialization, fire, preflight, and launch fail. Initialization fail is the first fault that can occur in the system, these faults are less extreme. The initialization sequence includes connection fail, countdown clock fail and hazard lights fail. Any of these can be mitigated with a retry before an abort command is issued. The fire fault tree sequence contains the most critical failures that could result in explosion of the system. These failures are LO2, helium and LH2 fail. Preflight fail are the faults that can occur before a launch command is issued. Preflight fail includes battery check, initialize fuel and battery switch fail. Launch is the final set of faults that can occur after the launch command has been issued. It includes environmental control system ECS and preflight fail. ECS includes the air conditioning failures and Nitro Purge failures. Preflight fail includes the Instrument, cryotesting and chill down failures. The fire, prelaunch and launch faults must be mitigated with an abort to protect the payload.

Four launch failure occurrences are described as four FTs, one FT for each failure. The FT in Figure 4.21 shows what causes the initialization failure of the launch vehicle, the FT in Figure 4.22 shows what can cause a fire and possible explosion. The pre-flight failure is illustrated by the FT in the Figure 4.23, and the launch failure is shown in Figure 4.24. These FTs will be integrated in the behavioral model shown in Figure 4.20. The mitigation actions for this system is to abort. Therefore, mitigations are not applicable.

Initialization fail FT and the event description are as follows:

- Connection fail: The first step in the launch sequence requires that a connection is made between the launch vehicle, upper stage, launch platform and

ground system. This connection consists of Ethernet cable to establish the ground network and 1553 cables for commanding and getting status from the launch vehicle. Failure for one of the networks to communicate would result in the launch being canceled or delayed. A retry action could be taken to attempt to establish the connection.

- CountDownClk fail:The launch vehicle and the ground system heavily rely on the countdown clock to synchronize time between them. If the countdown clock fails to start, pause or stop the result could fail to synchronize and cause a tank to be over/under filled and an explosion. If the fault were caught early on, the ground operator could retry to sync them or abort the launch.
- HazardLight fail: Hazard Lights are used for safety around a launch vehicle. They consist of flashing or strobe lights to warn people in the area to keep away. The launch should not be conducted with a failure in the safety light mechanism.

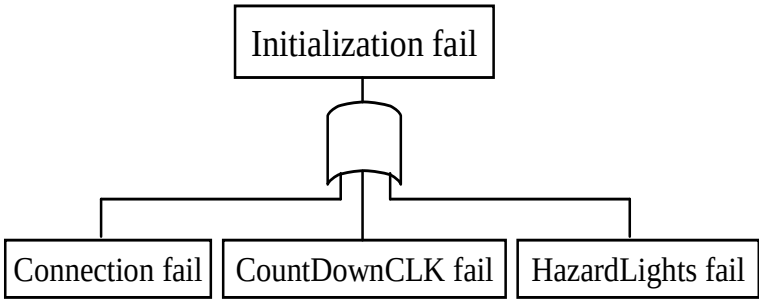

Figure 4.21: Initialization Fail FT

Fire fail FT and the event description are as follows:

- LO2 fail: Liquid oxygen is cryogenic liquid oxidizer propellant for a launch vehicle. It creates a high specific impulse. The launch vehicle tank is made of thin material which is filled with L02 to pressurize it. However, LO2 will

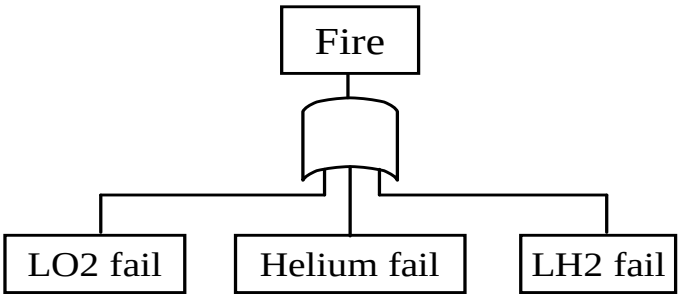

Figure 4.22: Fire Occurrence FT

boil off and must be replenished before launch. Liquid Oxygen is fed into the engine using valves. Faults associated with LO2 include: failure to pressurize, failure to top off tank, stuck valve, or defective structural integrity of the tank. The faults if not mitigated in time would result in a fire or explosion.

- Helium fail: Helium is used by the upper stage to purge fuel and pre-cool liquid hydrogen. A failure from helium would result in liquid oxygen overheating and an explosion of the system.
- LH2 fail: Liquid Hydrogen is the upper stage cryogenic rocket propellant. It has the lowest molecular weight of any substance and burns with extreme intensity. Liquid hydrogen creates the highest specific impulse. The faults associated with Liquid Hydrogen include, exposure to heat and leaking out of tank weld seams which would cause an explosion.

Pre-flight fail FT and the event description are as follows:

- BAChk fail: Battery checks are performed on the launch vehicle by the ground system. Batteries are tested for condition, state of charge is measured in volts, cell resistance is measured ohms, and a percent of life expectancy is evaluated. Faults include: bad condition, low voltage, low cell resistance and low life expectancy.

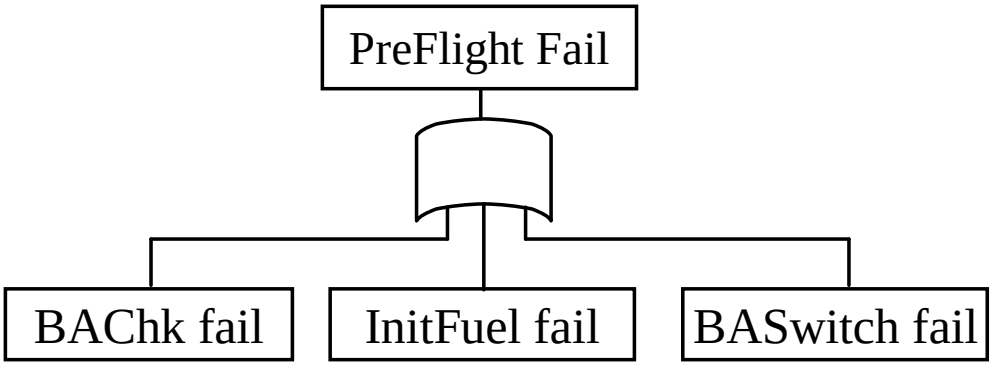

Figure 4.23: Preflight Fail FT

- InitFuel fail: Fuel Initialization is the process of preparing the booster LO2 system and the upper stage LH2 system. The fuel systems are prepared by locking the valves and measuring gas pressure. Faults include low fuel pressures or bad valves.
- BASwitch fail: Prior to launching, the ground system must switch the launch vehicle from external power to internal power. This is accomplished by switching the power to the internal batteries. Internal battery failures include failure to switch, bad battery condition, low voltage, low cell resistance and low life expectancy.

Launch fail FT and the event description are as follows:

- ACInit fail: Launch pad environmental control system air conditioning is initialized. The system fails when the air conditioning unit fails to power, or temperature is not within an acceptable range.
- NitroPurge fail: Launch pad environmental control system performs a nitrogen purge of the tanks prior to launch. Nitrogen is used to clean the tanks of impurities. It will also displace oxygen and reduce the risk of fire or oxidation. Faults that could occur are low nitrogen pressure or stuck valve.

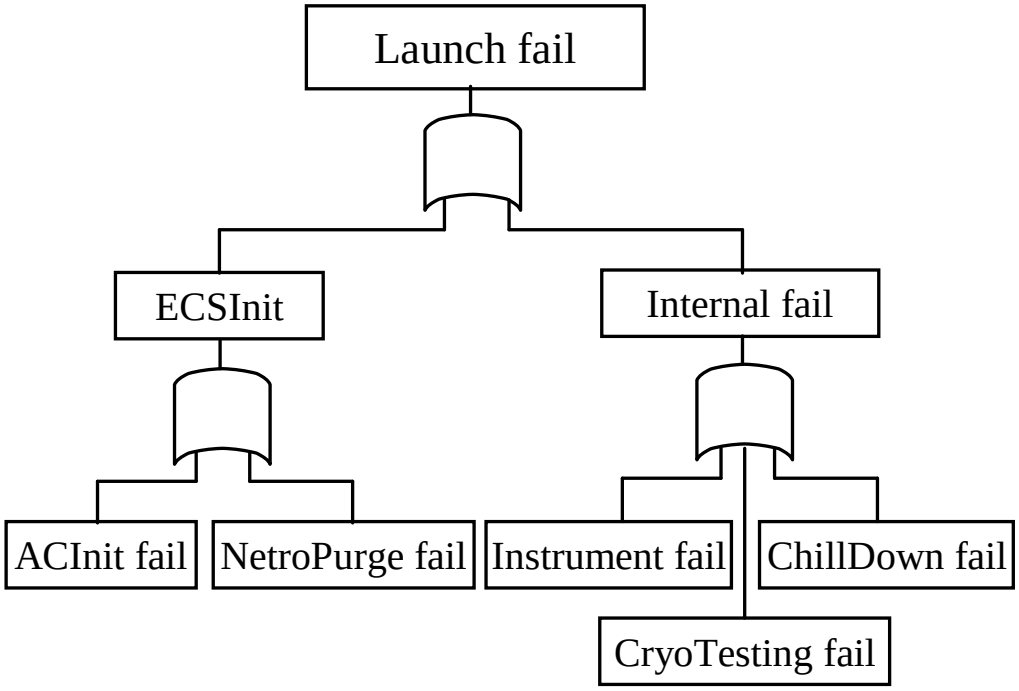

Figure 4.24: Launch Fail FT

- Instrument fail: Prior to launch, the vehicle's instrumentation is verified by running a self or BIT (built in test) Test, the self-test verifies the instrumentation is running properly and performs a check sum to ensure that the proper version of software is loaded. Instrumentation faults include self-test failure, checksum error or telemetry data error.
- ChillDown fail: The chilldown procedure is used to condition fuel lines to handle the extreme cold temperatures of the cryogenic fuel. Small amounts of fuel are released from the storage tanks into the lines the feed the vehicle. Failures include: low chilldown pressure or ruptured fuel line.
- CryoTesting fail: Cryotesting is used to determine if the vehicle will operate under extreme temperatures. This demonstration fills and drains the tanks several times. Failures include: failure to pressurize tanks and valve failure.

4.2.2.3 Compatibility Transformation Step

At this step we create *Bclass* and *Fclass* for failure related entities and combine the related classes according to the compatibility transformation procedure. At this step we create *Bclass* and *Fclass*. In this example, four FTs will be integrated to the behavioral model. We start with the left most leaf node of the FT in Figure 4.21. The leaf node *Connection fail* of the fault tree in Figure 4.21 is related to the entity *Network Connection*. Therefore, according to the compatibility transformation rules, since the attribute of the *Bclass BNetworkConnection* and the *Fclass FConnection* are the same, they are combined in the *BFclass BFConnection*.

Next, we take its sibling, the *CountDownCLK fail* which is represented as *FCountDownCLK* class. This *Fclass* is related to the entity *Countdown Clock* at the behavioral model which is represented as *BCountDownClock*. Therefore, they are combined into *BFCountDownCLK* class. The third leaf node in this FT is the *HazardLights fail*. This leaf node event is related to the *HazardLights On*. Therefore, we combine their related *Bclass* and *Fclass* into *BFHazardLights*. Notice that, here the values of the attributes are different, therefore, we need to include a *BAttribute* (*Bstate*) from the *BhazardLights* and *FAttribute* from the *FHazardLights* into the *BFHazardLights*.

Next, we do the compatibility transformation for the second FT Figure 4.22. We start with the left most leaf node which is the event *LO2 fail* that is represented as *FLO2Chk*. This event is related to the entity *LO2Chk* which is represented as *BLO2Chk*. Since the attributes of these classes are the same, we combine them into *BFLO2* as shown in Figure 4.28. The next event to transform is the *Helium fail*. It is related to the *HeliumChk* entity and both have the same attributes. Therefore, they are combined into *BFHelium* (Figure 4.29.) The next event in this FT is the

leaf node *LH2 fail*. It is related to the *LH2Chk* entity at the behavioral model. The *LH2Chk* and *LH2 fail* are represented as *BLH2Chk* and *FLH2* respectively. Since these events have the same attributes, they are combined in *BFLH2*.

Having finished all leaf node events in the FT in Figure 4.22, we start with the left most leaf node event of the *PreFlight fail* FT (Figure 4.23), which is *BAChk fail* that is represented as *FBAChk*. It is related to the entity *BatteryChk* which is represented as *BBatteryChk* class. These classes are combined in *BFBatteryChk* class (*cf* Figure 4.31.) *InitiFuel fail*, represented as *FInitFuel*, is related to *Initiate-Fueling* entity which is represented as *BInitiateFueling*. As shown in Figure 4.32, these two classes are combined in *BFInitFuel*. Figure 4.33 shows the combination of the event *BASwitch fail* and *IntBattery*. These two events are represented in *BInternalBattery* and *FBSwitch* classes respectively.

Next we analyze the fault tree for launch fail (Figure 4.24) starts with the left most leaf node event which is *ACInit fail*. This event is represented as *FACInition* class and is related to the entity *Air Conditioning* which is also represented as *BAirCondition*. The attributes of these classes are the same so they are combined in *BFACInitiation* class as shown in Figure 4.34. The next leaf node event is *NitroPurge fail* which is related to the entity *NitrogenPurge* at the behavioral model. The *NetroPurge fail* is represented as *FNitrogenPurge* class and the *BNitrogenPurge* is represented as *NitrogenPurge* class. These two classes are combined in *BFNitrogenPurge* as shown in Figure 4.35. Next, we take the event *Instrument fail*. This event is related to the *INSTChk* entity. They are represented as *FInstrument* and *BINSTChk* respectively and combined into *BFInstrument* as illustrated in Figure 4.36.

The event *CryoTesting fail* is transformed next. This event is represented in *FCryoTesting* and is related to the *CryoTesting* entity which is also represented as

BCryoTesting class. The combination of these two classes is the *BFCryoTesting* class can be seen in Figure 4.37. Finally, we transform the event *ChillDown fail* to be compatible with the entity *ChillDown*. They are represented as *FChilldown* class and *BChilldown* class and are combined in *BFChilldown* class as Figure 4.38 shows.

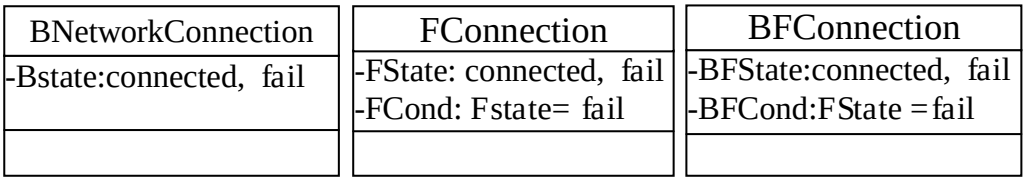

Figure 4.25: Network Connection Class

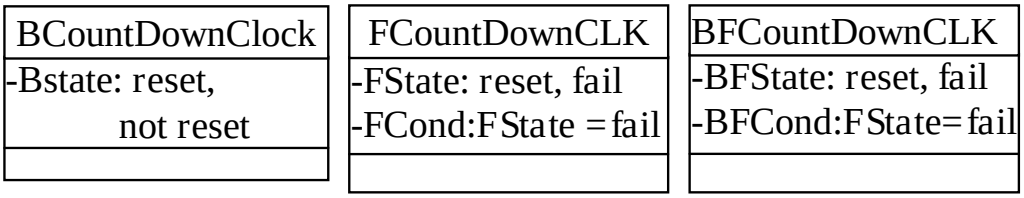

Figure 4.26: Countdown Clock Class

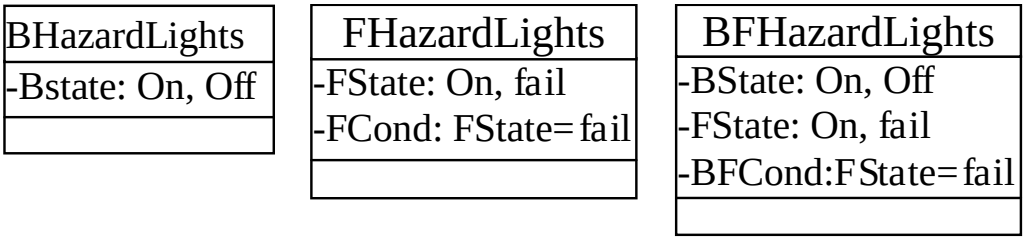

Figure 4.27: Hazard Lights Class

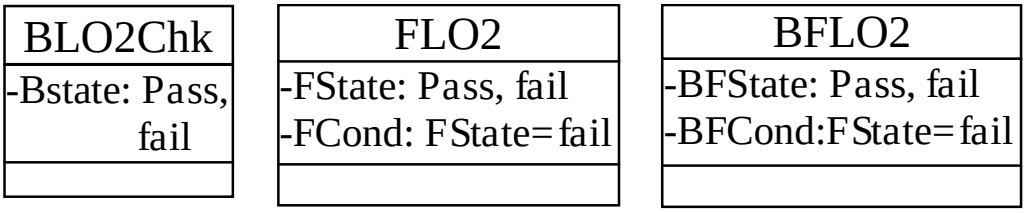

Figure 4.28: LO2 Class

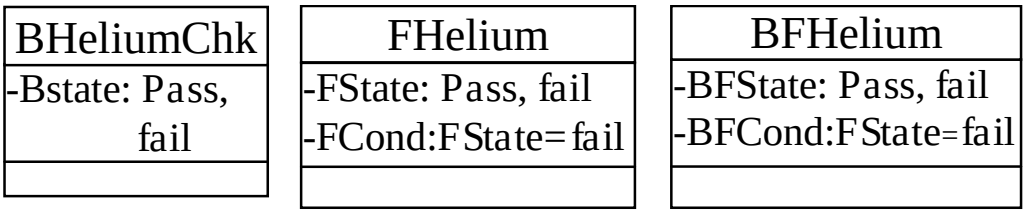

Figure 4.29: Helium Class

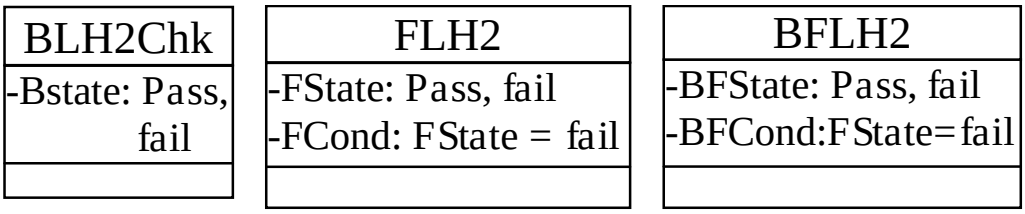

Figure 4.30: LH2 Class

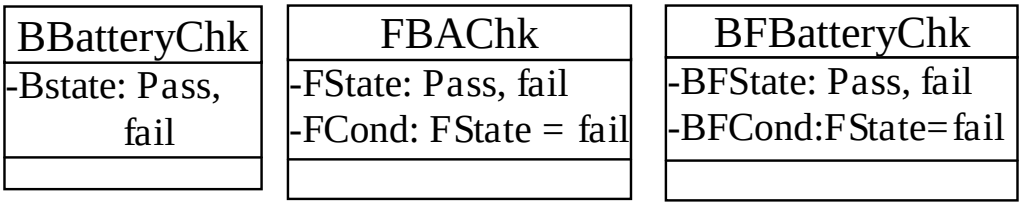

Figure 4.31: Battery Class

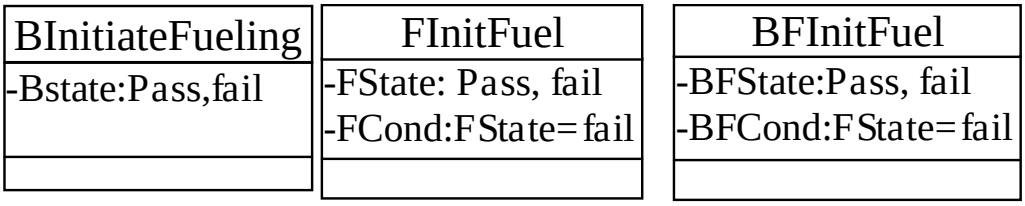

Figure 4.32: Initiating Fueling Class

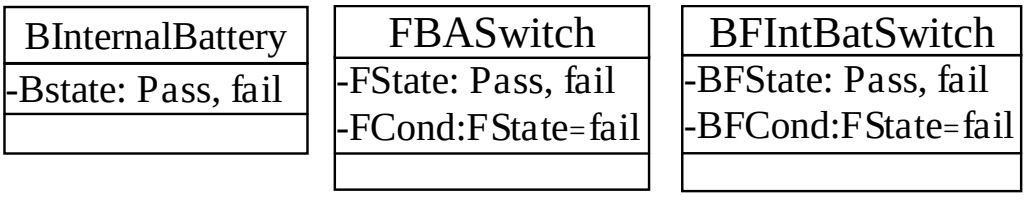

Figure 4.33: Battery Switching Class

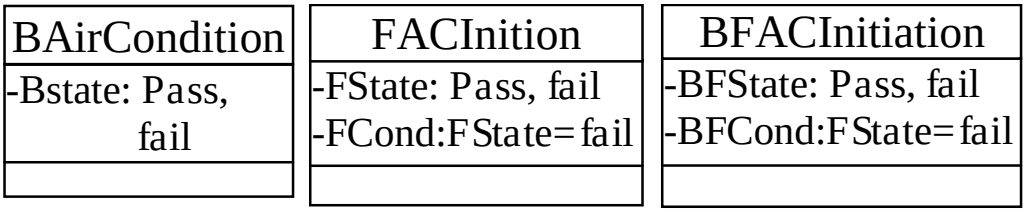

Figure 4.34: Air Conditioning Initiation Class

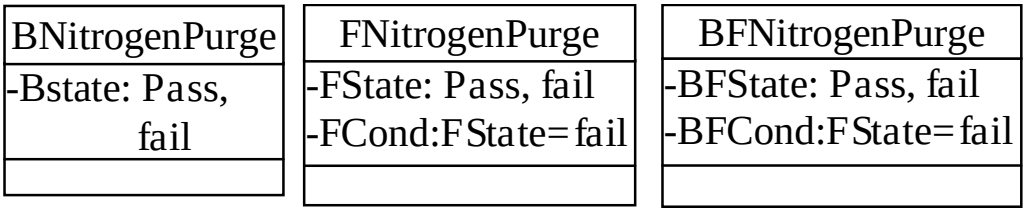

Figure 4.35: Nitrogen Class

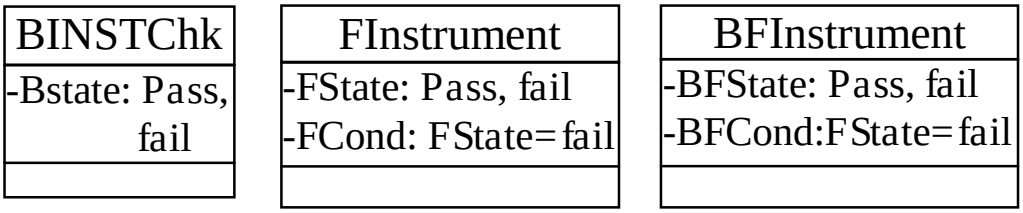

Figure 4.36: Instruments Class

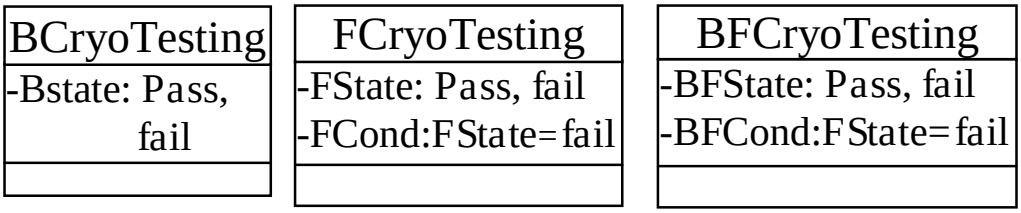

Figure 4.37: Cryo Class

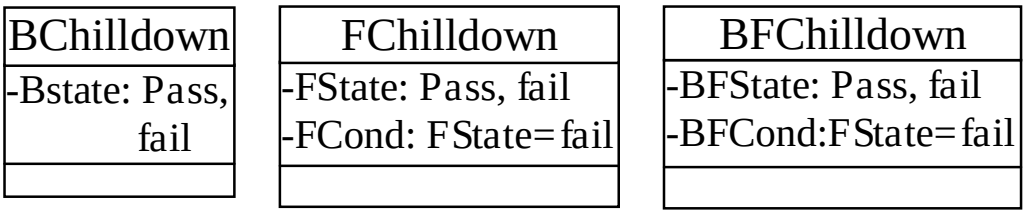

Figure 4.38: Chill Down Class

After the compatibility transformation procedure is finished, the fault tree of the initialization failure Figure (4.21) is represented as: FT′ = (∨,(*BFConnection.BFCond*,*BFCountDownCLK.BFCond*,*BFHazardLight. BFCond*)).

The FT in Figure 4.22 is presented as: FT′ = (∨,(*BFLO2.BFCond*,*BFHeluim.BFCond*,*LH2.BFCond*))

The FT in Figure 4.23 is presented as: FT′ = (∨,(*BFBatteryChk.BFCond*,*BFInitFuel.BFCond*,*BFIntBatSwitch.BFCond*))

The FT in Figure 4.24 is presented as: FT′ = (∨,(∨,*BFACInitiation.BFCond*,*BFNitrogenPurge.BFCond*),(∨,(*BFInstrument .BFCond*,*BFCryoTesting.BFCond*,*BFChilldown.BFCond*)))

4.2.2.4 Fault Tree Transformation

The fault CEFSM is constructed according to a tree postorder traversal. Each FT is read gate by gate starting from the root node until we reach the leftmost leaf node. The transformation starts with the leftmost leaf of the FT. The events are described in terms of class diagram states and events as shown in the compatibility transformation step. We start with the Initialization fail FT of Figure 4.21. We traverse this FT from the root to the left most leaf node, the *connection fail*. Since this is a leaf node, we give it an Event ID and the Gate ID, and insert it in the Event-Gate table. Each event and the Gate ID are assigned a unique sequential ID according to their appearance in the table. The next event is the *CountDownCLK fail* as expressed in the condition determined by the compatibility step and the third

is *HazardLights fail*. These events are shown in Table 4.9. This FT contains only one gate and its GCEFSM can be seen in Figure 4.39.

Table 4.9: Event-Gate Table after Transforming FT in Figure 4.21

Event name & attribute	Event ID	Gate ID
BFConnection.BFCond	e_{B1}	G_1
BFCountDownCLK.BFCond	e_{B2}	G_1
BFHazardLight.BFCond	e_{B3}	G_1

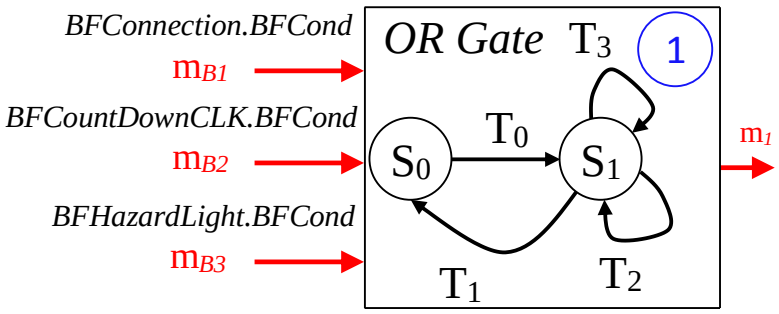

Figure 4.39: GCEFSM for the FT in Figure 4.21

The next FT to transform into GCEFSM is the fire occurrence FT, Figure 4.22. We start with the left most leaf node which is *LO2 fail*. Since it is a leaf node, we give it an Event ID and the Gate ID, and insert it in the Event-Gate table. Then we take its siblings from left to right. The next sibling is the *Helium fail* event, give it and event ID and insert it in the Event-Gate table and then take the last sibling and do the same thing. At this point, all the leaf nodes of the this FT are processed, we create the gate and give it a gate ID. Figure 4.40 shows the GCEFSM for this FT and Table 4.10 shows the Event-Gate table after the transformation of this FT.

The pre-flight fail FT in Figure 4.23 is then transformed following the same procedure. The first leaf node is *BAChk fail*. We give this event an event ID and inset it into the Event-Gate table with the gate ID that this event is linked to. We

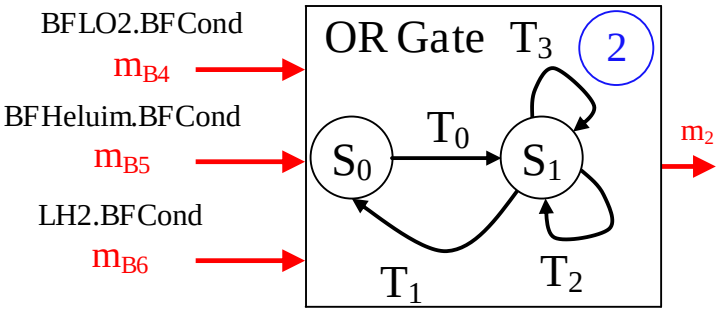

Figure 4.40: GCEFSM for Fire Occurrence FT in Figure 4.22

Table 4.10: Event-Gate Table after Transforming FT in Figure 4.22

Event name & attribute	Event ID	Gate ID
BFConnection.BFCond	e_{B1}	G_1
BFCountDownCLK.BFCond	e_{B2}	G_1
BFHazardLight.BFCond	e_{B3}	G_1
BFLO2.BFCond	e_{B4}	G_2
BFHeluim.BFCond	e_{B5}	G_2
LH2.BFCond	e_{B6}	G_2

take its sibling, *InitFuel fail* and we give it an event ID and inset it into the Event-Gate table. We do the same thing with the last event in this FT is the *BASwitch fail* and then we create the gate. The GCEFSM of this FT is shown in Figure 4.41 and the Event-Gate table is shown in Table 4.11.

The *Launch fail* FT is then transformed to an equivalent GCEFSM. The leaf node *ACInit fail* is read first, given an event ID and inserted into the table. Second, the event *NitroPurge fail* is read and given an Event ID and inserted into the Event-Gate table. Then the GCEFSM OR gate is created. This step is shown in Figure 4.42.

Next, we take the leaf node *Instrument fail*, *CryoTesting fail*, and *ChillDown fail* one after another and we take the same action for each one. At this point, all

Table 4.11: Event-Gate Table after Transforming FT in Figure 4.23

Event name & attribute	Event ID	Gate ID
BFConnection.BFCond	e_{B1}	G_1
BFCountDownCLK.BFCond	e_{B2}	G_1
BFHazardLight.BFCond	e_{B3}	G_1
BFLO2.BFCond	e_{B4}	G_2
BFHeluim.BFCond	e_{B5}	G_2
LH2.BFCond	e_{B6}	G_2
BFBatteryChk.BFCond	e_{B7}	G_3
BFInitFuel.BFCond	e_{B8}	G_3
BFIntBatSwitch.BFCond	e_{B9}	G_3

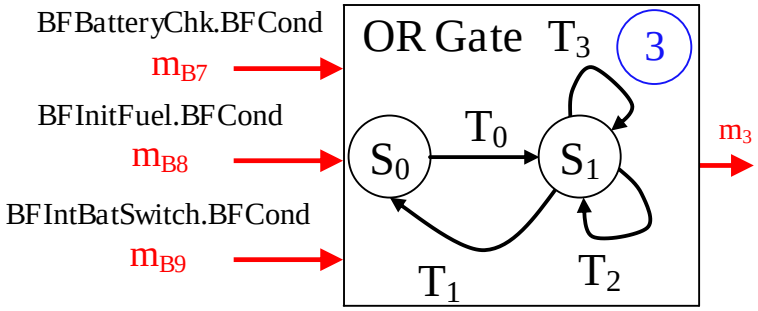

Figure 4.41: GCEFSM for the Preflight Failure FT in Figure 4.23

the leaf nodes of this FT are read and all the related gates are transformed into GCEFSMs. Figure 4.43 shows the GCEFSM for this gate and Table 4.12 shows the contents of the Event-Gate table at this point. Next the gate is transformed into GCEFSM.

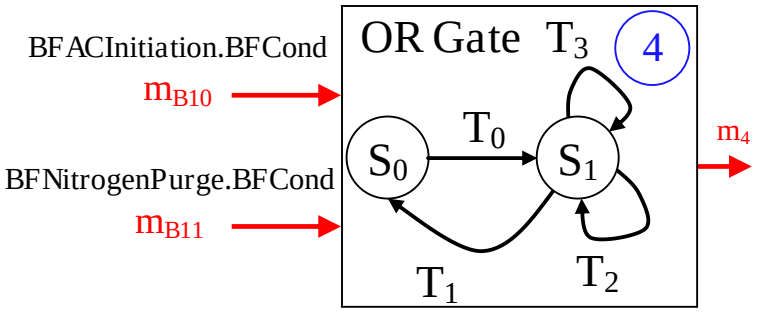

Figure 4.42: GCEFSM for an OR Gate in Figure 4.24

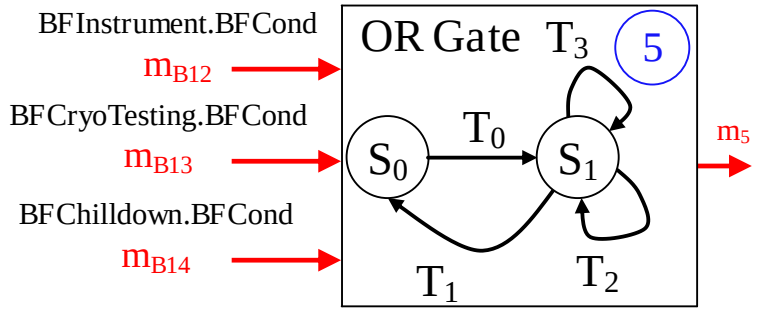

Figure 4.43: GCEFSM for the Second OR Gate in Figure 4.24

Then we take the gate at the upper level of this FT. This gate is an OR gate. We transform it and assign the events from the lower level gates. Figure 4.44 shows the whole *flight fail* GCEFSM.

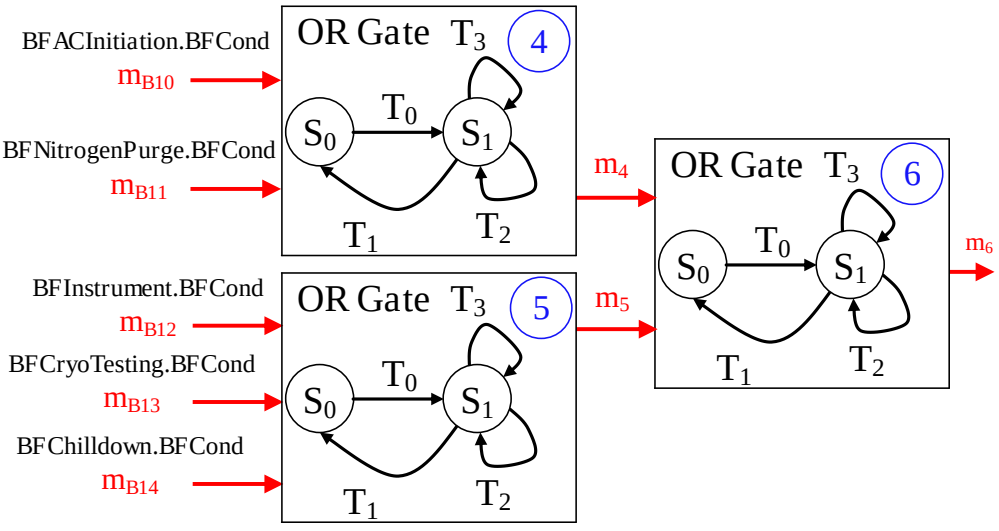

Figure 4.44: GCEFSM for Flight Fail FT in Figure 4.24

Table 4.12: Event-Gate Table after Transforming FT in Figure 4.24

Event name & attribute	Event ID	Gate ID
BFConnection.BFCond	e_{B1}	G_1
BFCountDownCLK.BFCond	e_{B2}	G_1
BFHazardLight.BFCond	e_{B3}	G_1
BFLO2.BFCond	e_{B4}	G_2
BFHeluim.BFCond	e_{B5}	G_2
LH2.BFCond	e_{B6}	G_2
BFBatteryChk.BFCond	e_{B7}	G_3
BFInitFuel.BFCond	e_{B8}	G_3
BFIntBatSwitch.BFCond	e_{B9}	G_3
BFACInitiation.BFCond	e_{B10}	G_4
BFNitrogenPurge.BFCond	e_{B11}	G_4
BFInstrument.BFCond	e_{B12}	G_5
BFCryoTesting.BFCond	e_{B13}	G_5
BFChilldown.BFCond	e_{B14}	G_5

4.2.2.5 Model Integration

After all fault trees are transformed to GCEFSMs, we start integrating them into the behavioral model. At this point, every message in the BM contains an event name that is related to an event in one of the fault trees. We check the class diagram and the Event-Gate table to find the event ID and the gate ID for the event. These event ID and gate ID are inserted into the message at the BM. The event "NetworkConnection fail" in the message mf_2 is represented in the class diagram as *BFConnection.BFCond*. This event is looked up inside the the event-gate table to obtain its event ID (e_{B1}) and the gate ID (G_1) the message is sent to. The message is modified as (m_{B1}, e_{B1}, G_1).

The event HazardLights fail in the message next message mf_2 is represented as *BFCountDownCLK.BFCond* in the class diagram. This event is looked up in the event-gate table to obtain its event ID and the gate ID for the gate that receives this event. They are e_{B2} and G_1 respectively. The message is modified as (m_{B2}, e_{B2}, G_1). The next message to be modified is the message that carries the event "CountDownCLK fail". This event is represented as *BFCountDownCLK.BFCond*. The event ID and the gate ID for this event are e_{B3} and G_1 respectively. The modified message will look like (m_{B3}, e_{B3}, G_1). These events happen to be for the same FT and this FT has only these three events as leaf nodes which means that the first FT is integrated.

The next event to check is "ACError" in the message mf_9. This event is represented in the class diagram as *BFACInitiation.BFCond*. This event is looked up in the event-gate table to obtain its event ID and the gate ID this event is an input to which are e_{B10} and G_4. The message will be (m_{B10}, e_{B10}, G_4). The next event from the BM is "fuelcheck Fail" in the message mf_{12}. This event is represented as *BFNitrogenPurge.BFCond*. Its event ID and gate ID are looked up in the event-gate table. This message will be modified as (m_{B11}, e_{B11}, G_4).

The event "LO2 fail" in the message mf_{14} which is represented as *BFLO2. BFCond* is looked up in the event-gate table for the event ID and the gate ID. the message will be modified to be (m_{B4}, e_{B4},G_2). The "Helium fail" in the message mf_{16} is looked up in the event-gate table to obtain its ID and the gate ID for the gate this event is an input to. This event is represented as *BFHelium.BFCond*. The message is modified to be (m_{B5},e_{B5},G_2). The event "LH2fail" in the message mf_{18} at the behavioral model is taken next. According to the compatibility transformation, this event is represented as *LH2.BFCond* and from the event-gate table its ID is e_{B6} and the gate ID this event is sent to is G_2. Therefore, the message is modified as

(m_{B6}, e_{B6}, G_2). The next event from the begavioral model to check is "Instrufail" in the message mf_{21}. This event is represented as *BFInstrument.BFCond*. Its event ID and gate ID are looked up in the event-gate table and they are e_{B12}, G_5. Therefore the message is modified as (m_{B12}, e_{B12}, G_5).

The integration procedure continues for all remaining messages from the behavioral model. These messages are: the message mf_{23} carrying the event "cryoTestingfail" becomes (m_{B13}, e_{B13}, G_5), the message mf_{25} carrying the event "ChillDownfail" becomes (m_{B14}, e_{B14}, G_5), the message mf_{27} carrying the event "Batteryfail" becomes (m_{B7}, e_{B7}, G_3), the message mf_{29} carrying the event "initiateFueling fail" becomes (m_{B8}, e_{B8}, G_3), and the message mf_{32} carrying the event InternalBatteryfail becomes (m_{B9}, e_{B9}, G_3). Ones all the messages are assigned to their GCEFSMs destinations, The behavioral model and the fault trees are integrated. Figure 4.45 shows the integrated model ICEFSM.

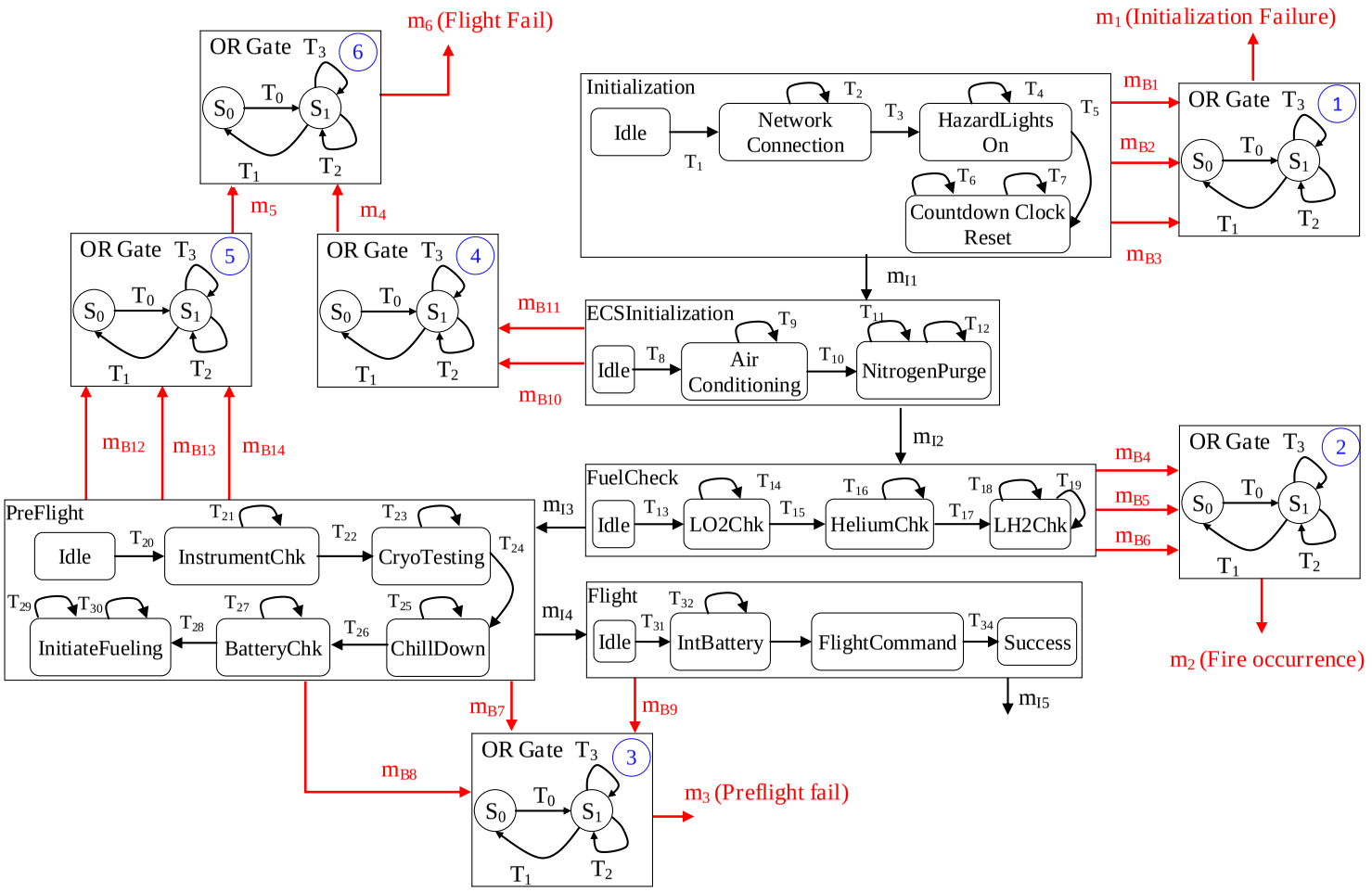

Figure 4.45: ICEFSM Model for a Launch System

Table 4.13: ICEFSM model for a launch System Transitions

- **T_1:**(Idle,[startSequence=True],startConnection)/(NetworkConnection,-)
- **T_2:**(NetworkConnection,[ConnectionConfirmed=false|timeout>= 30000])/(NetworkConnection,-,send(m_{B1}))
- **T_3:**(NetworkConnection,[ConnectionConfirmed=True],TurnLightsOn) /(HozardLightsOn,-,-)
- **T_4:**(HazardsightsOn, [AllHazardLighsOn = false],)/(HazardLightsOn,-, send(m_{B2}))
- **T_5:**(nazardLightsOn, [AllHazardLighsOn = true], ResetClock)/(Count DownClockReset ,-,-)
- **T_6:**(CountDoenclockRwset, [ClkError = true],)/(CountDownclock Reset,-,send(m_{I1}))
- **T_7:**(CountDownclocLReset, [ClkError = false],)/(CountDownclock Reset, -, send(m_{B3}))
- **T_8:**(Idle,get(m_{I1}))/(AirConditioning,-,-)
- **T_9:**(AirConditioning,[ACError=true],purge)/(AirConditioning,-, send(m_{B10}))
- **T_{10}:**(AirConditioning,[ACError=false],purge)/(NitrogenPurge,-,-)
- **T_{11}:**(NitrogenPurge,[ECSError=false])/(NitrogenPurge,-,send(m_{B11}))
- **T_{12}:**(NitrogenPurge,[ECSError=ture])/(NitrogenPurge,-,send(m_{I2}))
- **T_{13}:**(Idle,,get(m_2"fuelcheck")/(LO2Chk,-,-)
- **T_{14}:**(LO2Chk,[LO2leak=true|LO2PressureOk=flase])/(LO2Chk, send(m_{B4}))
- **T_{15}:**(LO2Chk, [LO2leak = false & LO2PressureOK = true])/

Continued on next page

Continued from previous page

(HeliumChk,-,-)

- $\mathbf{T}_{16}$**:**(HeliumChk,[Helkimleak = true|HeliumPressureOK = false])/ (HeliumChk, -, send(m_{B5}))
- $\mathbf{T}_{17}$**:**(HeliumChk,[Heliumleak = false&HeliumPressureOK = true])/ (LH2Chk,-,-)
- $\mathbf{T}_{18}$**:**(LH2Chk,[LH2leak = true|LH2PressureOk != false])/(LH2Chk, send(m_{B6}))
- $\mathbf{T}_{19}$**:**(LH2Chk,[LH2leak = false & LH2PressureOK = true])/(LH2Chk ,-,send(m_{I3}))
- $\mathbf{T}_{20}$**:**(Idle,,get(m3"PreFlight")/(INSTChk,-,-)
- $\mathbf{T}_{21}$**:**(INSTChk,[ChkcksumOK = false | LaunchConductCommOk = false])/(INSTChk,send(m_{B12}))
- $\mathbf{T}_{22}$**:**(INSTChk,[ChecksumOk = true & LaunchConductCommOk = true])/(CryoTesting, -,-)
- $\mathbf{T}_{23}$**:**(CryoTesting, [IntTempOK = false | IntPressureOk = false]) / (CryoTesting, send(m_{B13}))
- $\mathbf{T}_{24}$**:**(CryoTesting, [IntTempOK = true & IntPressureOk = true])/ (ChillDown,-,-)
- $\mathbf{T}_{25}$**:**(ChillDown,[IntTempOK = false | InterPssurOK = false])/ (ChillDown,-, send(m_{B14})
- $\mathbf{T}_{26}$**:**(ChillDown,[IntTemIOK = true & IntPressurOK = true])/ (BatteryChk,-,-)
- $\mathbf{T}_{27}$**:**(BatteryChk,[BatBeryPresent = false | PowerLevelOK = false | BatteryLifeOK = false])/(BatteryChk,-,send(m_{B7}))
- $\mathbf{T}_{28}$**:**(BatteryChk, [BatteryPresent = true & PowerLevelOK = true &

Continued on next page

Continued from previous page

ButteryLifeOK = true])/(InitiateFueling,-,-)

- **T_{29}:**(InitiateFueling, [TankPressureOK = false | FuelLevelOK = false | TankTempOK = false])/(InitiateFueling,send(m_{B8}))
- **T_{30}:**(InitiateFueling, [TankPressureOK = true & FuelLevelOK = true & TankTempOK = true])/(InitiateFueling, send(m_{I4})
- **T_{31}:**(Idle,,gyt(m_4" Flight")/(InternalBattery,-,-)
- **T_{32}:**(InternalBattery,[SwitchToBatteryOK=false|PowerLevelOK = false])/(InternaeBattery,send(m_{B9})
- **T_{33}:**(InternalBattery,[SwitchToBatteryOK=true&PowerLevel OK= true])/(FlightCommand,-,-)
- **T_{34}:**(FlightCommand,[StartFlight=true],StartFlight)/(Success,-, send(m_{I5}))

- m_{B1}**:**(m_{B1}, e_{B1}, G_1)
- m_{B2}**:**(m_{B2}, e_{B2}, G_1)
- m_{B3}**:**(m_{B3}, e_{B3}, G_1)
- m_{B4}**:**(m_{B4}, e_{B4}, G_2)
- m_{B5}**:**(m_{B5}, e_{B5}, G_2)
- m_{B6}**:**(m_{B6}, e_{B6}, G_2)
- m_{B7}**:**(m_{B7}, e_{B7}, G_3)
- m_{B8}**:**(m_{B8}, e_{B8}, G_3)
- m_{B9}**:**(m_{B9}, e_{B9}, G_3)
- m_{B10}**:**(m_{B10}, e_{B10}, G_4)
- m_{B11}**:**(m_{B11}, e_{B11}, G_4)
- m_{B12}**:**(m_{B12}, e_{B12}, G_5)
- m_{B13}**:**(m_{B13}, e_{B13}, G_5)
- m_{B14}**:**(m_{B14}, e_{B14}, G_5)

Table 4.14: Aerospace Launch System Test Paths

- Initialization[Idle$\xrightarrow{T_1}$NetworkConnection$\xrightarrow{T_3}$HazardLightsOn$\xrightarrow{T_5}$Count DownClockReset]
- ECSInitialization[idle$\xrightarrow{T_8}$AirConditioning$\xrightarrow{T_{10}}$NitrogenPurge$\xrightarrow{T_{11}}$Nitrogen Purge]
- FuelCheck[Idle$\xrightarrow{T_{13}}$LO2Chk$\xrightarrow{T_{15}}$HeliumChk$\xrightarrow{T_{17}}$LH2Chk$\xrightarrow{T_{18}}$LH2Chk]
- PreFlight[Idle$\xrightarrow{m_{20}}$InstrumentChk]$\xrightarrow{m_{b12}}$ S_9 $\xrightarrow{m_5}$ S_{11} $\xrightarrow{T_{51}}$ S_{12}
- PreFlight[Idle$\xrightarrow{m_{20}}$InstrumentChk$\xrightarrow{T_{22}}$CryoTesting$\xrightarrow{T_{24}}$ChillDown$\xrightarrow{T_{26}}$ BatteryChk$\xrightarrow{T_{28}}$InitiaFueling$\xrightarrow{T_{29}}$InitiateFueling]
- PreFlight[Idle$\xrightarrow{m_{20}}$InstrumentChk$\xrightarrow{T_{22}}$CryoTesting$\xrightarrow{T_{24}}$ChillDown$\xrightarrow{T_{26}}$ BatteryChk]$\xrightarrow{m_{B8}}$ S_5 $\xrightarrow{T_{39}}$ S_6
- Initialization[Idle$\xrightarrow{T_1}$NetworkConnection$\xrightarrow{T_2}$NetworkConnection]$\xrightarrow{m_{B1}}$ S0$\xrightarrow{T_{31}}$ S_1
- Initialization[Idle$\xrightarrow{T_1}$NetworkConnection$\xrightarrow{T_3}$HazardLightsOn]$\xrightarrow{m_{B2}}$ S_0 $\xrightarrow{T_{31}}$ S_1
- Initialization[Idle$\xrightarrow{T_1}$NetworkConnection$\xrightarrow{T_3}$HazardLightsOn$\xrightarrow{T_5}$Count DownClockReset]$\xrightarrow{m_{B2}}$ S_0 $\xrightarrow{T_{31}}$ S_1
- Initialization[Idle$\xrightarrow{T_1}$NetworkConnection$\xrightarrow{T_3}$HazardLightsOn$\xrightarrow{T_4}$Hazard LightsOn$\xrightarrow{T_5}$CountDown Clock Reset]
- Initialization[Idle$\xrightarrow{T_1}$NetworkConnection$\xrightarrow{T_3}$HazardLightsOn$\xrightarrow{T_5}$Count

Continued on next page

Continued from previous page

DownClockReset] $\xrightarrow{m_1}$ ECSInitialization[Idle $\xrightarrow{T_8}$ AirConditioning $\xrightarrow{T_{10}}$ NitrogenPurge]

- ECSInitialization[Idle $\xrightarrow{T_8}$ AirConditioning $\xrightarrow{T_9}$ Air conditioning $\xrightarrow{T_{10}}$ NitrogenPurge]
- ECSInitialization[Idle $\xrightarrow{T_8}$ AirConditioning $\xrightarrow{T_{10}}$ NitrogenPurge] $\xrightarrow{m_{B11}}$ S_7 $\xrightarrow{m_4}$ S_{11} $\xrightarrow{T_{51}}$ S_{12}
- ECSInitialization[Idle $\xrightarrow{T_8}$ AirConditioning $\xrightarrow{T_{10}}$ NitrogenPurge] $\xrightarrow{m_2}$ FuelChk[idle $\xrightarrow{T_{13}}$ LO2Chk $\xrightarrow{m_{15}}$ HeliumChk] $\xrightarrow{m_{B5}}$ S_3
- FuelChk[Idle $\xrightarrow{T_{13}}$ LO2Chk $\xrightarrow{T_{14}}$ LO2Chk $\xrightarrow{T_{15}}$ HeliumChk] $\xrightarrow{m_{B5}}$ S_4
- FuelChk[Idle $\xrightarrow{T_{13}}$ LO2Chk $\xrightarrow{T_{15}}$ HeliumChk $\xrightarrow{T_{16}}$ HeliumChk] $\xrightarrow{m_{B5}}$ S_4
- FuelChk[Idle $\xrightarrow{m_{13}}$ LO2Chk $\xrightarrow{T_{15}}$ HeliumChk $\xrightarrow{T_{17}}$ LH2Chk] $\xrightarrow{m_{I3}}$ PreFlight[Idle $\xrightarrow{T_{20}}$ InstrumentChk] $\xrightarrow{m_{B12}}$ S_9 $\xrightarrow{m_5}$ S_{11} $\xrightarrow{T_{51}}$ S_{12}
- PreFlight[Idle $\xrightarrow{T_{20}}$ InstrumentChk $\xrightarrow{T_{22}}$ CryoTesting] $\xrightarrow{m_{B13}}$ S_9 $\xrightarrow{m_5}$ S_{11} $\xrightarrow{T_{51}}$ S_{12}
- PreFlight[Idle $\xrightarrow{T_{20}}$ InstrumentChk $\xrightarrow{T_{22}}$ CryoTesting $\xrightarrow{T_{24}}$ ChillDown] $\xrightarrow{m_{B14}}$ S_9 $\xrightarrow{m_5}$ S_{11} $\xrightarrow{T_{51}}$ S_{12}
- PreFlight[Idle $\xrightarrow{m_{20}}$ InstrumentChk $\xrightarrow{T_{22}}$ CryoTesting $\xrightarrow{T_{23}}$ CryoTesting $\xrightarrow{T_{24}}$ ChillDown $\xrightarrow{T_{26}}$ BatteryChk $\xrightarrow{T_{28}}$ InitiaFueling]
- PreFlight[Idle $\xrightarrow{m_{20}}$ InstrumentChk $\xrightarrow{T_{22}}$ CryoTesting $\xrightarrow{T_{24}}$ ChillDown $\xrightarrow{T_{25}}$ ChillDown $\xrightarrow{T_{26}}$ BatteryChk $\xrightarrow{T_{28}}$ InitiaFueling]

Continued on next page

Continued from previous page

- PreFlight[Idle $\xrightarrow{m_{20}}$ InstrumentChk $\xrightarrow{T_{22}}$ CryoTesting $\xrightarrow{T_{24}}$ ChillDown $\xrightarrow{T_{26}}$ BatteryChk $\xrightarrow{T_{27}}$ BatteryChk $\xrightarrow{T_{28}}$ InitiaFueling]
- PreFlight[Idle $\xrightarrow{m_{20}}$ InstrumentChk $\xrightarrow{T_{22}}$ CryoTesting $\xrightarrow{T_{24}}$ ChillDown $\xrightarrow{T_{26}}$ BatteryChk $\xrightarrow{T_{28}}$ InitiaFueling] $\xrightarrow{m_{B8}}$ S_5 $\xrightarrow{T_{39}}$ S_6
- PreFlight[Idle $\xrightarrow{m_{20}}$ InstrumentChk $\xrightarrow{T_{22}}$ CryoTesting $\xrightarrow{T_{24}}$ ChillDown $\xrightarrow{T_{26}}$ BatteryChk $\xrightarrow{T_{28}}$ InitiaFueling] $\xrightarrow{m_4}$ Flight[idle $\xrightarrow{T_{31}}$ IntBattery $\xrightarrow{T_{33}}$ FlightCommand $\xrightarrow{T_{34}}$ Success]
- Flight[Idle $\xrightarrow{T_{31}}$ IntBattery $\xrightarrow{T_{32}}$ IntBattery $\xrightarrow{T_{33}}$ FlightCommand $\xrightarrow{T_{34}}$ Success]

The transformed system shown in Figure 4.45 forms a graph to which suitable coverage criteria can be applied. The FT gates that are directly connected to the behavioral model receive messages from the behavioral model and act accordingly. The messages m_1 to m_5 represent the global transitions between the GCEFSMs for the FTs, while m_{I1} to m_{I5} represent the messages between the components of the behavioral model and m_{B1} to m_{B14} represent the communicating messages between the behavioral and fault models. If we apply the algorithm in [68] to the graph in Figure 4.45 by imposing the edge coverage criteria, we obtain the test paths shown in Table 4.14. Note that this approach forces a proper prioritization if the tests are executed in order, i.e. there is no need for extra test prioritization rules.

The difference between our approach and those that use statecharts such as [127, 40, 82] is that our approach is used to explicitly model systems (with communication edges) where the behavior process and the failure process intersect. Therefore, paths can be produced. These paths can be used for feasibility testing and planning for mitigation actions, and mitigation testing. It is also possible to manipulate sensor

values and create failure events during system testing. Moreover, in our approach, different levels of details can be used for different testing purposes. For example, if we want to test the system, we can look at every GCEFSM as a unit and not worry about the GCEFSMs' internal details (transitions and states) since we know how they behave.

When we compared the number of states and transitions produced by our integrated approach with those of [127] on this aerospace launch system example, we found that our ICEFSM contains 41 states and 117 transitions whereas the EFSM model of [127] will contain at least 4316 states and 8335 transitions.

4.3 ICEFSM as Part of an End-to-End testing Methodology

The End-to-End testing[2] [6] methodology shown in Figure 4.46 consists of two phases. The first phase integrates the behavioral and fault model to produce test cases. The second phase construct the safety mitigation tests based on the weaving rules and some coverage criteria.

4.3.1 Test Generation Process

The test generation process is shown in Figure 4.46. It uses the behavioral model (BM) and a FT to generate test cases. The approach consists of two phases. The first phase generates failures and determines in which behavioral states specific failures may occur (failure applicability). Phase 1 starts with a compatibility transformation step for the Fault Tree wrt the behavioral model. The FT produced from this step is transformed into gate CEFSMs (GCEFSMs) according to transformation rules. Then, the model integration step integrates the GCEFSM with the behavioral model (BM) according to the integration rules. The resultant model is the integrated communicating extended finite state machine (ICEFSM). Any of a number of existing test case generation methods can use this model to generate test cases based on test criteria (IC). This is the approach described in Section 3 of this dissertation.

The second phase generates safety mitigation tests. We construct the behavioral test suite (BT) from the behavioral model (BM) using behavioral test criteria (BC). From the test paths generated from the integrated model ICEFSM, we construct

[2]This portion of the work is done jointly with Mrs. Salwa Elakeili.

the applicability matrix. Then, we apply test coverage criteria (the paper suggests C1-C4) to the behavioral test suite and failure types. Based on the applicability of failures in specific behavioral states, we select states in existing test paths representing behavioral tests (positions in a test path) and combine them with applicable failures(e) to systematically create failure situations for which we test proper mitigation. The mitigation tests are generated from the mitigation model (MM) based on mitigation coverage criteria (MC). After that, we construct the safety mitigation tests (SMTs) by combining the mitigation tests (MTs) with the behavioral tests at the failure position (p) according to weaving rules (WRs). We describe each phase in more detail in the following subsections.

4.3.2 Phase1: Generate Failures and Failure Applicability

In this phase, we propose an integration of the behavioral model with a fault model to take advantage of the two for testing. The test generation process in Figure 4.46 uses the behavioral model and a FT to generate test cases. It starts with the compatibility transformation step. The FT′ produced from this step is transformed into gate CEFSMs (GCEFSMs) according to the transformation rules. Then, the model integration step integrates the GCEFSM with the behavioral model (BM) according to the integration rules. The resultant model is the Integrated Communicating Extended Finite State Machine ICEFSM. Test case generation methods can use this model to generate test cases based on test criteria (IC) that lead to failure. A shorter version of the technique described here was originally developed in [52]. In [52, 54, 53] it was applied to multiple fault trees related to an aerospace launch system. However, it did not consider test generation for proper failure mitigation.

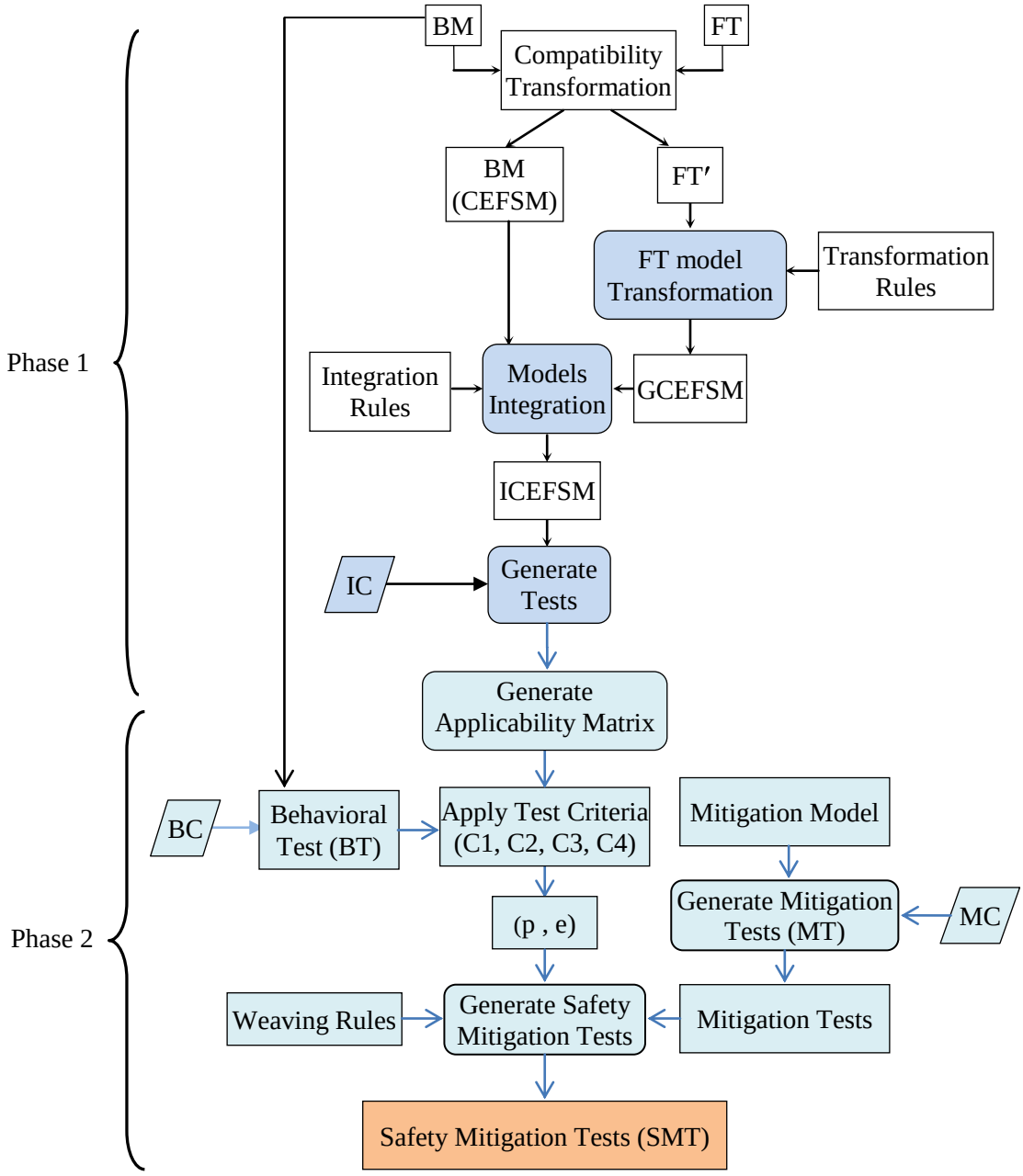

Figure 4.46: End-To-End Test Generation Process

4.3.3 Construction of the Applicability Matrix

The applicability matrix is a two dimensional array. Each column represents a specific behavioral state s ∈ S and each row is a specific failure type e ($1 \leq e \leq |E|$).

$$A(i,j) = \begin{cases} 1 & \text{if } failure\ type\ j\ can\ occur\ in\ state\ s_i, \\ 0 & \text{if } otherwise \end{cases}$$

Phase 1 determines whether a failure of type j can occur in state s_i or not. The applicability matrix is then constructed from the test paths obtained from phase 1 according to the construction procedure shown in Figure 4.47

Procedure ConstructionOfApplicabilityMatrix()

Inputs:

- Set of test paths (R) from ICEFSM.
- Failure types table (Q).

Output:

- Applicability Matrix AM(i,j).

Begin

For all the paths r_i in R take r_i one-by-one **{**

If (path $r_i \in$ R contribute to failure**)**

Then {

Obtain failure name (W_i) from Q

Check W_i in Q

If (found**)** //it mean that it is a failure not an event

then{

For every behavior state $s_i \in r_i$

Assign "1" to AM (f_j, s_i)

Check the reminder test paths $r'_j \in$ R that *don't* contribute to failure

If ($s_0 \in r_i == s'_0 \in r'_j$**)**

then

For all $s'_i \in r'_i$

Assign "1" to AM (f_j, s'_I)

}

else

// this is a normal event

}

}

And assign "0" to the reminder of AM

End

Figure 4.47: Applicability Matrix Construction Procedure

4.3.4 Phase2: Generate Safety Mitigation Tests

Our goal in this phase is to provide an MBT approach to test proper mitigation of safety failures in SCSs. Andrews et al. [8] describe this approach for generating a safety mitigation test suite. However, they use an EFSM instead of a CEFSM and these are not able to model the interaction of the failure process and the behavioral process and merely assume that a particular failure can be generated when the system is in a given behavioral state. However the process described below is substantially Mrs. Elakeili's work and described here to demonstrate how the end-to-end process works.

Mitigation models describe mitigation patterns associated with a fault. Mitigation test criteria describe required coverage. Mitigation test paths are then generated and woven into the behavioral test similar to aspect oriented modeling [129]. Weaving rules describe how a mitigation test path is woven into the original behavioral test. Phase 2 in figure 4.46 summaries how to construct safety mitigation tests (SMTs). The safety critical testing process has the following steps:

- Construct a behavioral test suite BT from the behavior model BM, using behavior test criteria BC.
- Construct mitigation test suites MT from mitigation models MM, using mitigation coverage criteria MC.
- Select positions of failure (p) in a test suite (BT), and type of failure (e) (failure scenarios). Select (p,e) using failure coverage criteria FC.
- Construct a safety mitigation test suite SMT using the behavioral test suite (BT), point of failure (p), type of failure (e) and mitigation test suite (MT) according to weaving rules (WR).

Next we need failure coverage criteria for failure scenarios. These are based on where in our behavioral test suites failure can occur and need to be tested. In other words, which positions p in the test suite need to be tested with which failure e? The test criteria specify coverage rules for selecting such (p,e) pairs.

Criteria 1: All combinations, i.e. all positions p, all applicable failure types e (test everything). This is clearly infeasible for all but the smallest models. It would require $|I| \times |F|$ pairs if A contains all "1"s.

Criteria 2: All unique nodes, all applicable failures. This only requires $\sum_{j=1}^{k} \sum_{i=1}^{|S|}$ (A(i,j)=1) combinations i.e. the number of one entries in the applicability matrix. When some nodes occur many times in a test suite only one needs to be selected by some scheme. This could lead to not testing failure recovery in all tests. A stronger test criterion is to require covering each test as well.

Criteria 3: All tests, all unique nodes, all applicable failures. Here we simply require that when unique nodes need to be covered they are selected from tests that have not been covered.

A weaker criterion is not to require covering all applicable failures for each selected position.

Criteria 4: All tests, all unique nodes, some failures (only one failure per position, but covering all failures). Some failure means that collectively all failures must be paired with a position at least once, but not with each selected position as in Criteria 3.

Example: This example shows the differences between the four types of coverage criteria for all combinations (p, e). Suppose we have a test suite that has three test paths T= $\{t_1,t_2,t_3\}$ where each test path contains a path.

$t_1 = \{s_1, s_2, s_3, s_4\}$, $t_2 = \{s_1, s_2, s_1, s_3, s_1, s_4\}$, $t_3 = \{s_2, s_4, s_3, s_2, s_3, s_4\}$.

CT= $t_1 \circ t_2 \ldots t_l = \{s_1, s_2, s_3, s_4, s_1, s_2, s_1, s_3, s_1, s_4, s_2, s_4, s_3, s_2, s_3, s_4\}$. I= [1,16].

Assume we have four failures F= $\{f_1, f_2, f_3, f_4\}$. $|F|$ =4 failures and we have four failure types E=[1,4]. The applicability matrix is shown in Table 4.15. Tables 4.16-4.19 show (p, e) pairs marked with "1" that, if selected, would collectively meet test criteria C1-C4 respectively.

Table 4.15: Applicability Matrix

F/S	s_1	s_2	s_3	s_4
f_1	1	1	1	1
f_2	1	0	0	1
f_3	1	1	1	0
f_4	1	1	1	1

Table 4.16: All Position, All Applicable Failures

F/CT	s_1	s_2	s_3	s_4	s_1	s_2	s_1	s_3	s_1	s_4	s_2	s_4	s_3	s_2	s_3	s_4
f_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f_2	1	0	0	1	1	0	1	0	1	1	0	1	0	0	0	1
f_3	1	1	1	0	1	1	1	1	1	0	1	0	1	1	1	0
f_4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 4.16 shows required combinations for criteria 1. This would need $|I| \times |F|$ minus the zeros entries (not applicable) in Table 4.16 (16 × 4 - 12 =64-12=52 pairs). For a tiny model with only 4 nodes and 4 failure types this is clearly too much.

For Criteria 2 consider Table 4.17. The options selected (marked 1) provide the desired coverage, but only test t_1 is used to fulfill this coverage. A total of 13 pairs is needed. According to Table 4.17 the following position-failure pairs (p,e) are

selected:{(1,1),(1,2),(1,3),(1,4),(2,1),(2,3),(2,4),(3,1),(3,3),(3,4),(4,1),(4,2),(4,4)}. A large portion of the test suite is unused. Random selection of nodes in I can improve this somewhat.

Criteria 3 requires using all tests. Table 4.18 shows an example of a set of position-

Table 4.17: All Unique Nodes, All Applicable Failures

F/CT	s_1 s_2 s_3 s_4 s_1 s_2 s_1 s_3 s_1 s_4 s_2 s_4 s_3 s_2 s_3 s_4
f_1	1 1 1 1
f_2	1 0 0 1
f_3	1 1 1 0
f_4	1 1 1 1

failure pairs (p, e) that fulfills this criterion {(1,1),(1,2),(1,3),(1,4),(6,1),(6,3),(6,4),(10,1),(10,2),(10,4),(13,1),(13,3),(13,4)}. As before, 13 pairs are needed, but the selection of unique nodes is spread over all three tests. Criteria 4 does not require

Table 4.18: All Tests, All Unique Nodes, All Applicable Failures

F/CT	t_1				t_2						t_3					
	s_1	s_2	s_3	s_4	s_1	s_2	s_1	s_3	s_1	s_4	s_2	s_4	s_3	s_2	s_3	s_4
f_1	1					1				1			1			
f_2	1									1						
f_3	1					1							1			
f_4	1					1				1			1			

that all failures be applied at every selected position although each failure must be selected at least once. Table 4.19 shows an example of selecting position-failure pairs (p, e). This is the weakest criterion, since it only requires selecting each failure at least once and each unique node at least once. The four position-failure pairs in Table 4.19 that fulfill this criterion are {(1,1),(6,3),(10,2),(13,4)}.

Table 4.19: All Tests, All Unique Nodes, Some Failures

F/CT	t_1				t_2						t_3					
	s_1	s_2	s_3	s_4	s_1	s_2	s_1	s_3	s_1	s_4	s_2	s_4	s_3	s_2	s_3	s_4
f_1	1															
f_2										1						
f_3						1										
f_4													1			

4.3.4.1 Generate Mitigation Test (MT)

Safety critical systems (SCSs) require mitigation of failures to prevent adverse effects. This can take a variety of actions. Mitigation patterns have been defined in [10][90] as follows:

1. **Rollback** brings the system back to a previous state before the failure occurred. A mitigation action may occur and the system may stop or proceed to re-execute the remainder of the test.
2. **Rollforward** mitigates the failure, fixes and proceeds.
3. **Try other alternatives** deals with decisions about which of several alternatives to pursue.
4. **Immediate(partial) fixing** when a failure is noted, an action is taken to deal with the problem that caused this failure prior to continuing with the remainder of the test.
5. **Deferred (partial) fixing** when a failure is noted, an action must be performed to record the situation and deal with the failure either partially or temporarily because handling the failure completely is not possible.

6. **Retry** when a failure is detected immediately after the execution of the activity causing the problem, an action is performed to solve the failure and then the activity that caused the problem is tried again.

7. **Compensate** means the system contains enough redundancy to allow a failure to be masked.

8. **Go to fail-safe state** a system is transferred into a mitigation state to avoid dangerous effects and stops.

These mitigation patterns can be expressed in the form of mitigation models. For example, try other alternatives is shown in Figure 4.48. Each failure f_i is associated

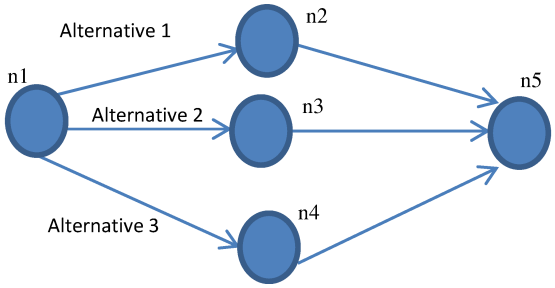

Figure 4.48: Try Other Alternatives: Mitigation Model

with a corresponding mitigation model MM_i where $i = 1, \ldots, k$. We assume that the models are of the same type as the behavioral model BM (e.g. an EFSM). Graph-based [5], mitigation criteria MC_i can be used to generate mitigation test paths $MT_i = \{mt_{i_1}, \ldots, mt_{i_{k_i}}\}$ for failure f_i. Figure 4.48 shows an example of a mitigation model of type "Try alternatives". Assuming MC as "edge coverage", the following three mitigation test paths fulfill MC: MT=$\{mt_1, mt_2, mt_3\}$ where

$mt_1 = \{n_1, n_2, n_5\}$, $mt_2 = \{n_1, n_3, n_5\}$, $mt_3 = \{n_1, n_4, n_5\}$

Mitigation models can be very small for some failures and the mitigation can be an "empty action". For example, if there is a rollback to state s_b with immediate stop,

the mitigation action only consists of adding a transition from s_b to s_f, the final state. Hence, mt=$\{s_b, s_f\}$. The weaving rule would specify what node to rollback to, in this case s_b. On the other hand, some mitigation models may consists of a full set of alternative behaviors that completely replace the remainder of the original test. We will illustrate this in the next section.

4.3.4.2 Generate Safety Mitigation Tests using Weaving Rules

Assume we have $t \in BT$, p $\in I$, $e \in E$ and mt $\in MT_e$. We now build a safety mitigation test smt $\in SMT$ using this information and the weaving rules $wr_e \in WR$ as follows:

- keep path represented by t until failure position p.
- apply failure of type e (f_e) in p.
- select appropriate mt $\in MT_e$.
- apply weaving rule wr_e to construct smt.

We now explain weaving rules more formally for each type of mitigation. Let t= $\{s_1 \ldots s_b \ldots node(\text{p}) \ldots s_f \ldots s_k\}$

1. **Fix**

   (a) **Compensate ((Partial) Fix and proceed)** mitigates a failure and continues with the remainder of the behavioral test. So,
   smt=$s_1 \ldots$.node(p)mtnode(p)$\ldots s_k$.
   mt may be zero, if mitigation does not require user involvement (inputs). See rule 4.

(b) **Go to fail-safe state (Fix and stop)** mitigates a failure and ignores the remainder of t: smt=$s_1 \ldots$.node(p)mt.

2. **Rollforward**

   (a) **Rollforward** mitigates the failure, and proceeds.

   smt =$s_1 \ldots$.node(p)$mts_f \ldots s_k$ where s_f is the node in t to which we rollforward. If only rollforward and no other actions are required mt is empty and smt=$s_1 \ldots$.node(p)$s_f \ldots s_k$.

   (b) **Deferred fixing.**

   If the failure can only be fixed after reaching the rollforward node s_f then smt becomes:

   smt = $s_1 \ldots$.node(p)$s_f mts_{f+1} \ldots s_k$.

   Note that further variants of this weaving rule can exist, like a state s_{df} between s_f and s_k at which the failure mitigation mt is inserted.

   t = $s_1 \ldots s_b \ldots$.node(p)$\ldots s_f \ldots s_{df} \ldots s_k$.

   smt= $s_1 \ldots$.node(p)$s_f \ldots s_{df}$mt$\ldots s_k$.

3. **Rollbackward**

   (a) **Rollbackward.**

   Apply mitigation path mt from point of failure and rollback to node before the failure occurred and continue with remainder of behavioral test.

   smt=$s_1 \ldots$.node(p)mt$s_b \ldots s_k$ where s_b is a node before node (p).

   (b) **Rollbackward and stop.**

   smt=$s_1 \ldots$.node(p)mts_b.

4. **Internal compensate (no user action required).** Test immediate system fix. For example, this can happen if a system switches to backup/redundant

sensors. To test this merely requires applying the failure and continuing to execute the original test t. In this case, we do not have to modify the original test at all (note that the assumption is that the system deals with the failure internally without any change in black-box behavior).

While weaving rules in this section are representative, they are not meant to be comprehensive. We expect that, over time, we may find some more or find that some are more common than others. The result of this step is the full safety mitigation test suite SMT.

4.4 End-To-End Case Study: Railroad Crossing Control System (RCCS)

In this section, we use the Railroad Crossing Control System (RCCS) to illustrate the whole End-to-End testing methodology.

4.4.1 Phase1: Generate Failures and Failure Applicability

4.4.1.1 Description of Railroad System

RCCS encompasses the following main components: train, railway track, sensors, gates, controller, and signal lights as shown in Figure 4.49. A depiction of each element is given below [101].

Train: A train is powered by a power supply. When the power is switched on, the train starts moving along the track when the metallic wheels of the train receive power. The train comes to a stop at the position where the power to the tracks is switched off.

Controller: The software that controls the general operation of the RCCS is stored in the memory of the controller. The controller continuously monitors the sensors and controls the gate actuators, track change lever, and the signal lights.

Sensors: Are used to detect the location of the train on the tracks. Two pairs of sensors detect the train position before and after the gates.

Gate: RCCS has two sets of gates on either side of the track layout. The gate receives signals from the controller. When it receives the command to lower the gate, the gate moves down and closes. When it receives the command to raise the gate, it moves up allowing the traffic to pass through.

Signal Lights: RCCS contains a warning light at the crossing area to indicate that the train is approaching when the light is on and there is no train otherwise.

A railroad crossing is an intersection where a railroad crosses a road or a path at the same level. Because of the safety concerns at the intersection, this system is intended to prevent normal traffic and people from using the intersection when a train approaches and crosses. Figure 4.49 depicts the behavioral model of a Railroad Crossing Control System (RCCS) as a Communicating Extended Finite State Machine (CEFSM) with one train and one track. The model specifies that gates are to be closed and warning lights are to be turned on when a train approaches, that they are to stay that way until the train is leaving. When the train is leaving, the gates are opened and the lights switched off. Gates stay open and lights are off while no train is approaching. The structure of the messages (Msg) in CEFSM is shown as in Table 4.20 (Msg_{id}, Event, CEFSM(MsgDestination))

Table 4.20: Structure of Messages

Msg_{id}	**Event**	**CEFSM**
Msg_1	Approaching=True	Controller " activated"
Msg_2	Crossing=True	Controller "monitor"
Msg_3	Leaving=False	Controller " deactivated"'
Msg_4	Activate=True	Gate "lower gate"
Msg_5	Monitor=False	Gate "raise gate"
Msg_6	Activate=True	Light "on"
Msg_7	Monitor=False	Light "off"

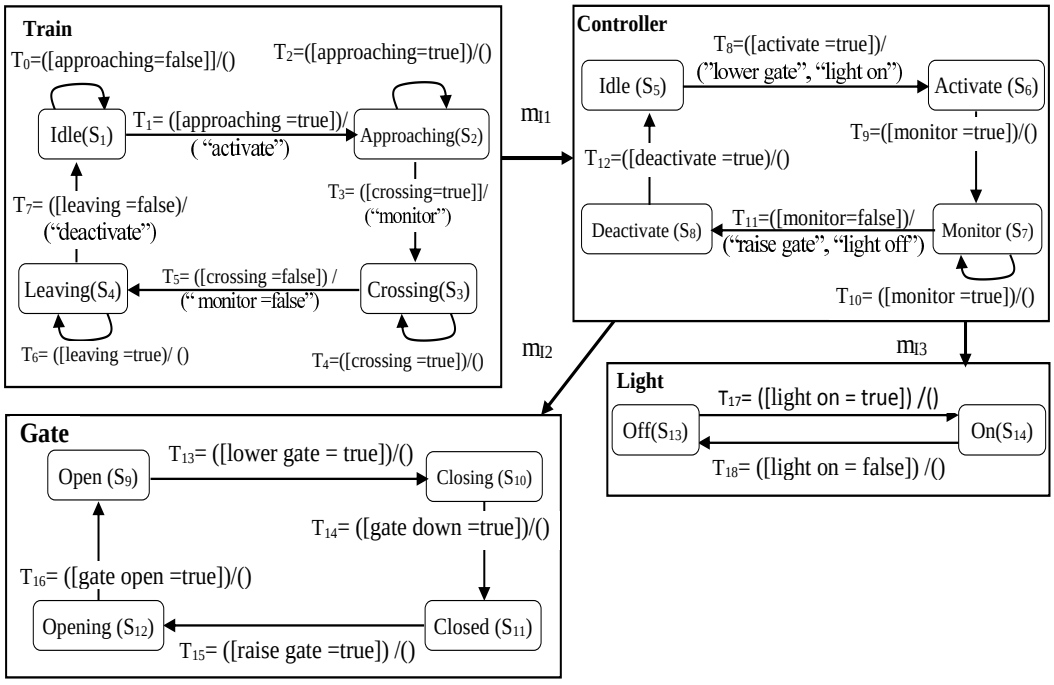

Figure 4.49: Railroad Crossing System Model

4.4.1.2 Railroad Crossing System Failure

Basic events description:

- Raise Gate is an action from the controller component to the gate component to open.
- Gate Opening means that the gate is in the opening state.
- Gate Open means that the gate is in the open state.
- Controller Fail means the controller has not received any message from the train (means sensor failed to detect the train) component and therefore it has not sent any message to the light to switch on or the gate to close.
- Controller Deactivated means that the controller has stopped monitoring the system as it received a message saying that a train has left the crossing area.
- Train Approaching indicates that a train is approaching when it hits a sensor.
- Train Crossing indicates that a train is in the crossing area when it hits a sensor.

This railroad crossing system example has one fault tree that describes a possible accident. This fault tree shows how some events or faults can cause an accident when they happen as the fault tree describes. For example, if the event *Train Approaching* is true and the event *Controller Fail* is true, the top event *accident* will be true, which means the hazard occurs. The fault tree shown in Figure 4.50 is described by

Accident = ($\wedge$,($\vee$,($\vee$, *Raise Gate*, *Gate Opening*),$\wedge$,(*Gate Open*, *Warning Light Off*)),$\vee$,(*Controller Fail*, *Controller Deactivate*)), ($\vee$,(*Train Approaching*, *Train Crossing*)))

Some events such as the *Train Crossing* and *Train Approaching* of the leaf nodes in the fault tree are normal events but they contribute to the accident when some

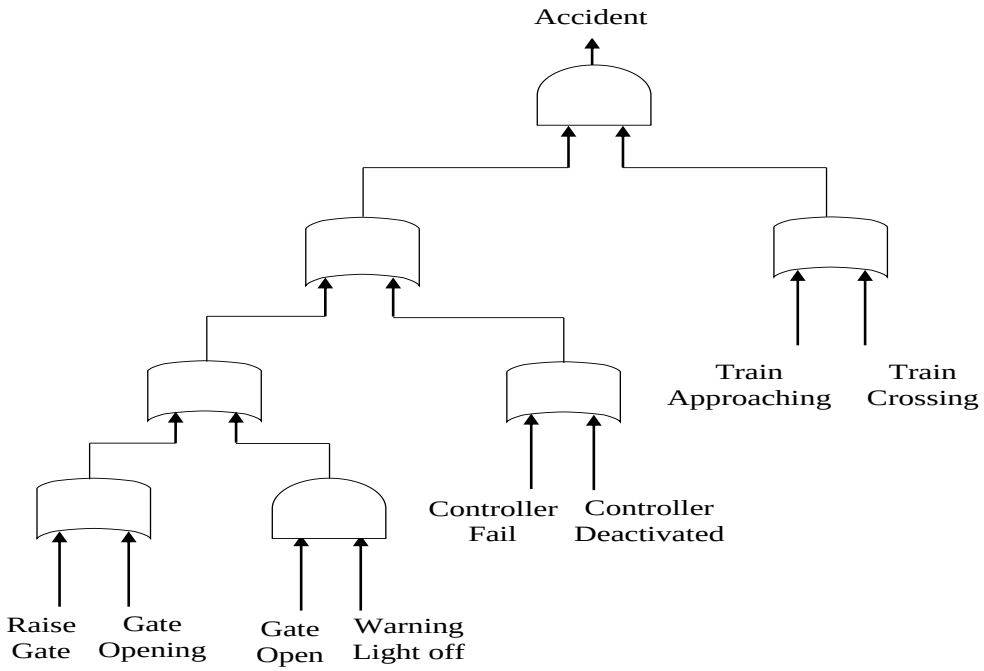

Figure 4.50: Fault Tree for Accident

other faults occur. For example, if the gate is open when the train is approaching, an accident may occur. At this point, the Failure Types Table shown in figure 4.21 contains only the *Failure IDs* and *Failure Types*.

Table 4.21: Failure Types Table

Failure ID	Failure Type	Node in FT′	Event ID	Gate ID	Message ID	Path ID
f_1	*Controller Fail*					
f_2	*Warning Light Fail*					
f_3	*Gate Stuck Open*					
f_4	*Controller deactivated*					

4.4.1.3 Compatibility Transformation Step

The first step is the compatibility transformation. At this step we create *Bclass* and *Fclass* for the failure related entities *BTrain*, *BController*,*BGate* and *BWarningLight* and combine the related classes according to the compatibility transformation procedure 3.1.3. These classes are shown in Figure 4.51, Figure 4.52, Figure 4.53,and Figure 4.54. The events in the FT are substituted with the combined attributes from the *BF classes* that are equivalent to these events. For example, the event *Raise Gate* in the FT is equivalent to *BFRaiseGate.BFEventCond* in FT′. The attributes of *BGate* and *FRaisGate* are combined in *BFRaisGate*. As we can see in Fig 4.53, the attribute *BState* belongs to the class *BGate* at the behavioral model and *FState* belongs to the *BFRaisGate* at the fault model.

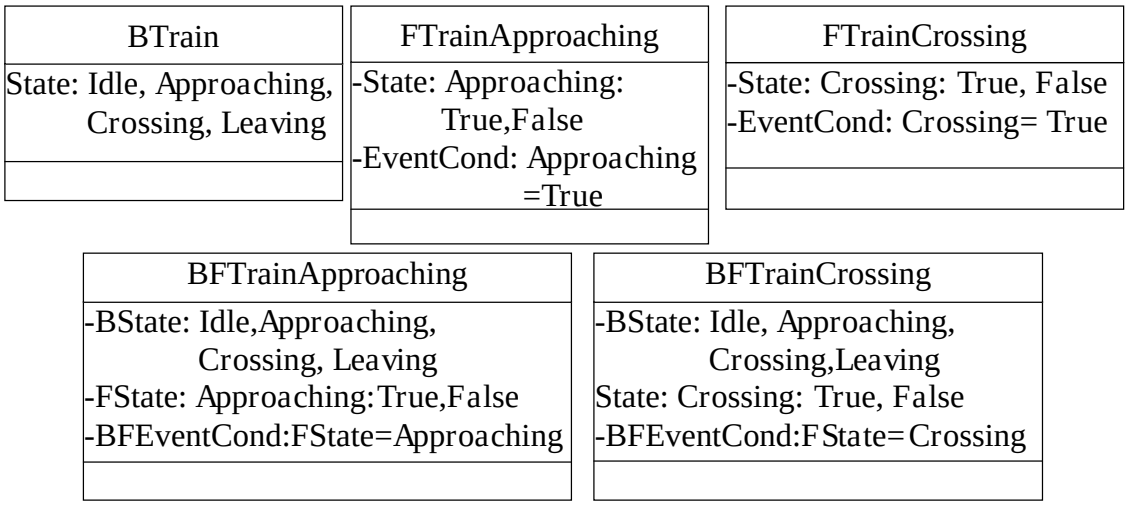

Figure 4.51: Train Approaching and Crossing Class

After the compatibility transformation procedure is finished, the complete fault tree *Accident* is represented as:

FT′= ($\wedge$, ($\vee$, ($\vee$, ($\vee$, *BFRaiseGate.BFEventCond*, *BFGateOpening.BF EventCond*) ,$\wedge$,(*BFGateOpen.BFEventCond*, *BFWarning.BFEventCond*)), $\vee$, (*BFControllerFail.*

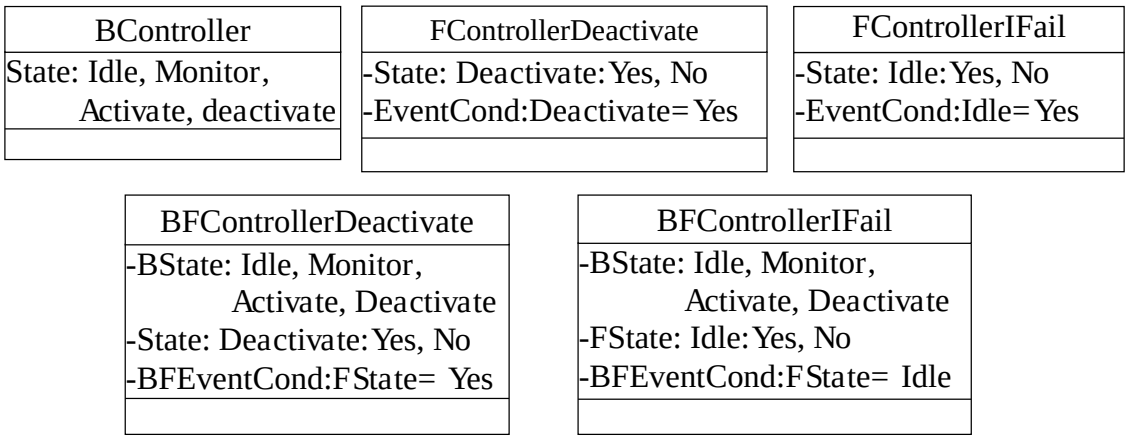

Figure 4.52: Train Controller Class Diagram

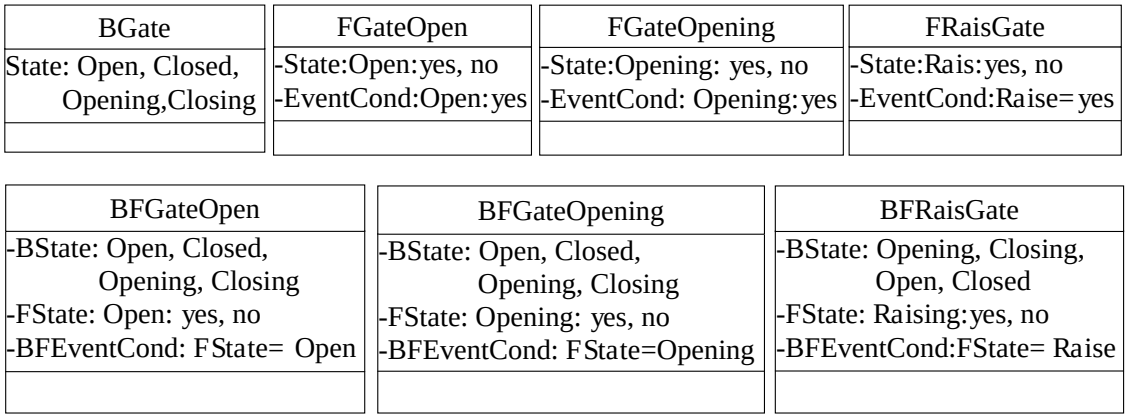

Figure 4.53: Gate Events Class Diagram

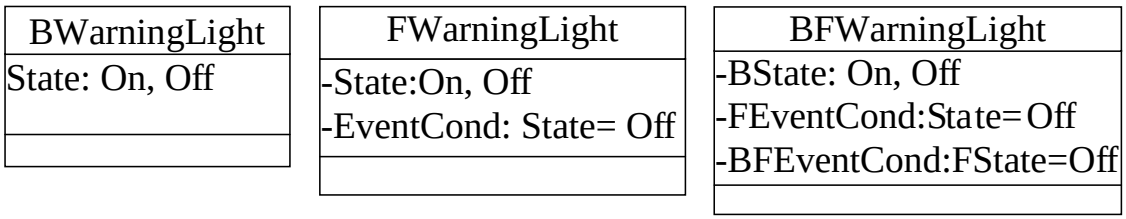

Figure 4.54: Warining Light Class Diagram

BFEventCond, *BFControllerDeactivate.BFEventCond*)), (∨, *BFTrainApproaching.BFEventCond*, *BFTrainCrossing.BFEventCond*))

At this step, the third column of the Failure Types Table is updated with the nodes in FT′ for every failure type. This is shown in Table 4.22.

4.4.1.4 Fault Tree Transformation

The fault CEFSM is constructed according to a tree postorder traversal. The FT is read gate by gate starting from the root node until we reach the leftmost leaf node. The transformation starts with the leftmost leaf of the FT which is in this example *Raise Gate*. The event is described in terms of class diagram as shown in Figure 4.53. The sibling of this event is *Gate Opening* which is also an event from the BM. The gate is constructed and given a number one because it is the first gate to construct in this FT. The message *id* should carry the same number as the gate. In this case the gate is given number one since it is the first gate to transform. The numbering of the internal transition is not important since each gate is an independent entity and no confusion will occur.

Next, we look for the right sibling of this gate which turns out to be an AND gate between two events. *Gate Open* and *Warning Light Off* The gate is shown in Figure 4.56. This gate is given number 2 since it is the second gate transformed. The next step it to transform the gate that combines these two gates and we give it number 3. The inputs to this gate are the output messages m_1 from gate 1 and message m_2 from gate 2. This gate is shown in Figure 4.57.

The next step is to transform gate number 4 and then gate number 5 which is the root for this subtree as shown in Figure 4.58. The next gate to transform is the sibling of gate number 5 which is given number 6 and then the root of these to subtrees is transformed which is given number 7. This step is shown in Figure 4.59

Table 4.22: Failure Types Table After Compatibility Transformation Step

Failure ID	Failure Type	Node in FT′	Event ID	Gate ID	Message ID	Path ID
f_1	*Controller Fail*	$BFConrollerFail.BFEventCond$				
f_2	*Warning Light Fail*	$BFWarning.BFEventCond$				
f_3	*Gate Stuck Open*	$BFGateOpen.BFEventCond$				
f_4	*Controller deactivated*	$BFConrollerDeavtivated.BFEventCond$				

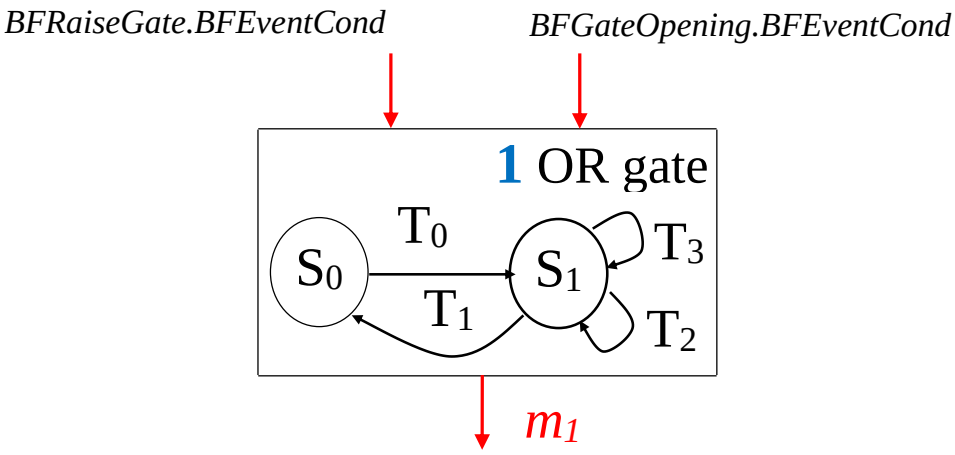

Figure 4.55: An OR Gate for the Left Most Event in the FT

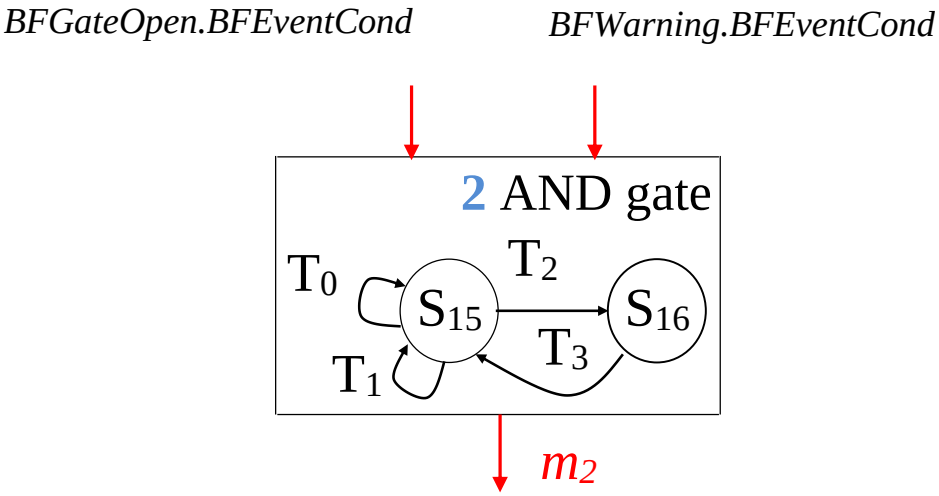

Figure 4.56: The Second Transformed Gate

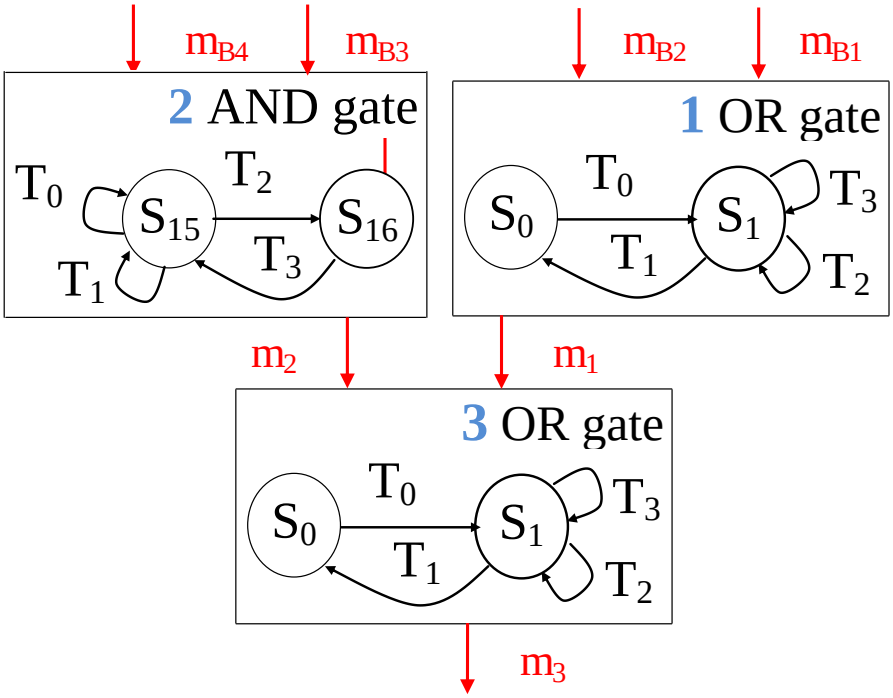

Figure 4.57: GCEFSM for Gate Number 3

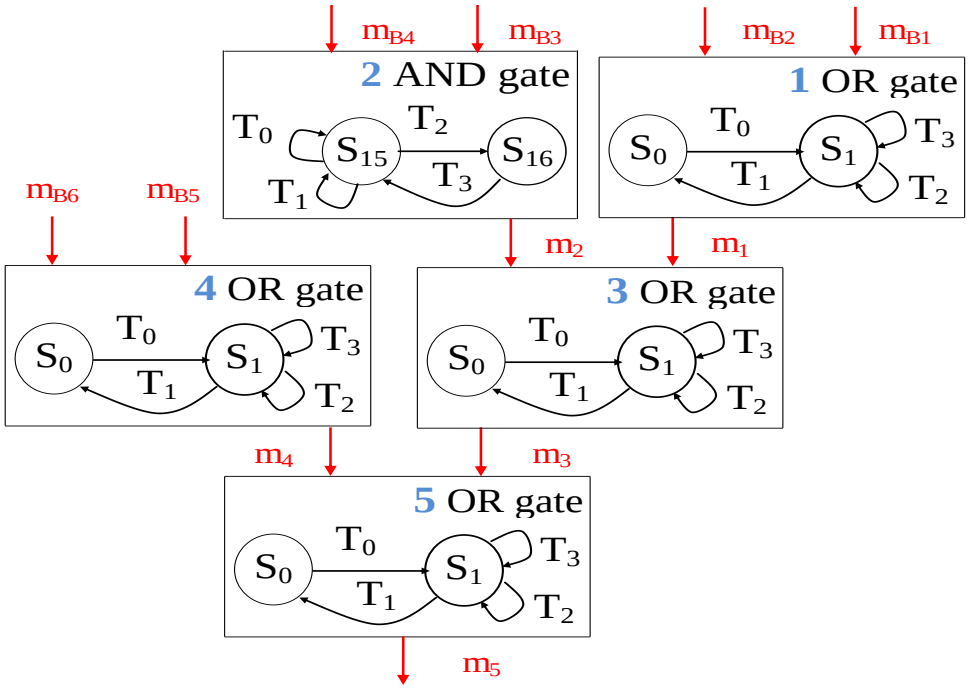

Figure 4.58: GCEFSM for Gates 1 to 5

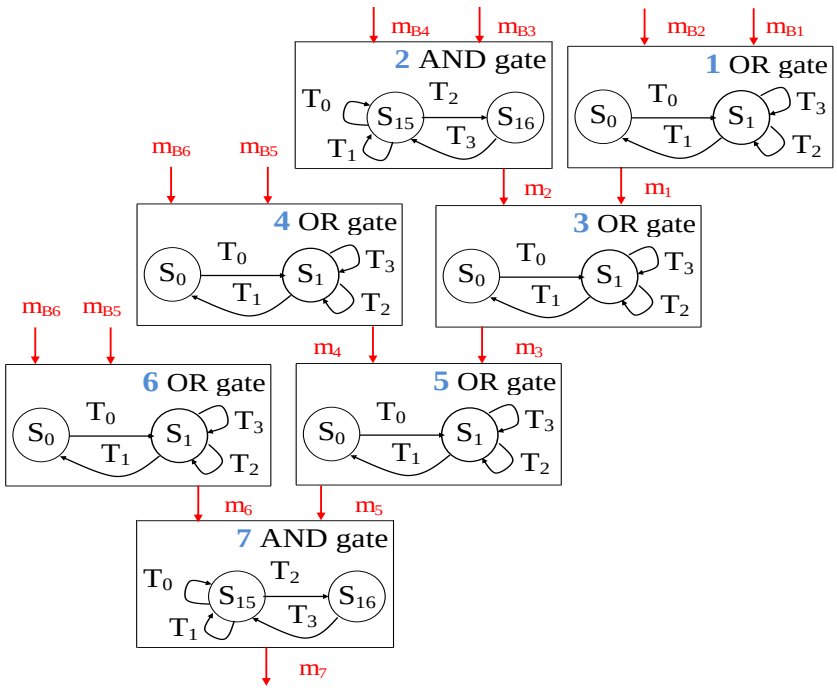

Figure 4.59: GCEFSMs for the Whole FT'

The event-gate table after the whole FT is transformed is shown in Figure 4.23. At this step, the forth and fifth columns of the Failure Types Table are updated with the *Event ID* and *Gate ID* for every failure type. This is shown in Table 4.24.

4.4.1.5 Model Integration

After the fault tree is transformed into GCEFSMs, we start integrating them into the behavioral model. At this point, every message in the BM contains an event name that is related to an event in one of the leaf nodes of the fault tree. We check the class diagram and the Event-Gate table to find the event ID and the gate ID for the event. These event IDs and gate IDs are inserted into the message at the BM. The event *Raise Gate* is represented in the class diagram as *BFRaiseGate.BFEventCond*. This event is looked up in the event-gate table to obtain its event ID (e_{B1}) and the gate ID (G_1) the message is sent to. The message in

Table 4.23: Event-Gate Table

Event name & attribute	Event ID	Gate ID
BFRaiseGate.BFEventCond	e_{B1}	G_1
BFGateOpening.BFEventCond	e_{B2}	G_1
BFGateOpen.BFEventCond	e_{B3}	G_2
BFWarning.BFEventCond	e_{B4}	G_3
BFControllerFail.BFEventCond	e_{B5}	G_4
BFControllerDeactivate.BFEventCond	e_{B6}	G_4
BFTrainApproaching.BFEventCond	e_{B7}	G_6
BFTrainCrossing.BFEventCond	e_{B8}	G_6

the BM is modified as (m_{B1}, e_{B1}, G_1). This procedure continues till all the messages in the BM are linked to the FM. At this step, the sixth column of the Failure Types Table is updated with the *Message ID* that carries the failure to the fault part of the model. This is shown in Table 4.25.

Figure 4.60 illustrates the RCCS transformed into an ICEFSM model. There are two connected models, the behavioral model and the FT model. The arrows between the CEFSMs represent the communicating messages between them. The transformed system shown in Figure 4.60 forms a graph to which suitable coverage criteria can be applied. The FT gates that are directly connected to the behavioral model receive messages from the behavioral model and act accordingly. The messages m_1 to m_7 represent the global transitions between the GCEFSMs for the FT part, while m_{I1} to m_{I3} represent the messages between the components of the behavioral model and m_{B1} to m_{B8} represent the communicating messages between the BM and FM.

Table 4.24: Failure Types Table After Model Transformation Step

Failure ID	Failure Type	Node in FT′	Event ID	Gate ID	Message ID	Path ID
f_1	*Controller Fail*	$BFConrollerFail.BFEventCond$	e_5	G_4		
f_2	*Warning Light Fail*	$BFWarning.BFEventCond$	e_4	G_2		
f_3	*Gate Stuck Open*	$BFGateOpen.BFEventCond$	e_3	G_2		
f_4	*Controller deactivated*	$BFConrollerDeavtivated.BFEventCond$	e_6	G_4		

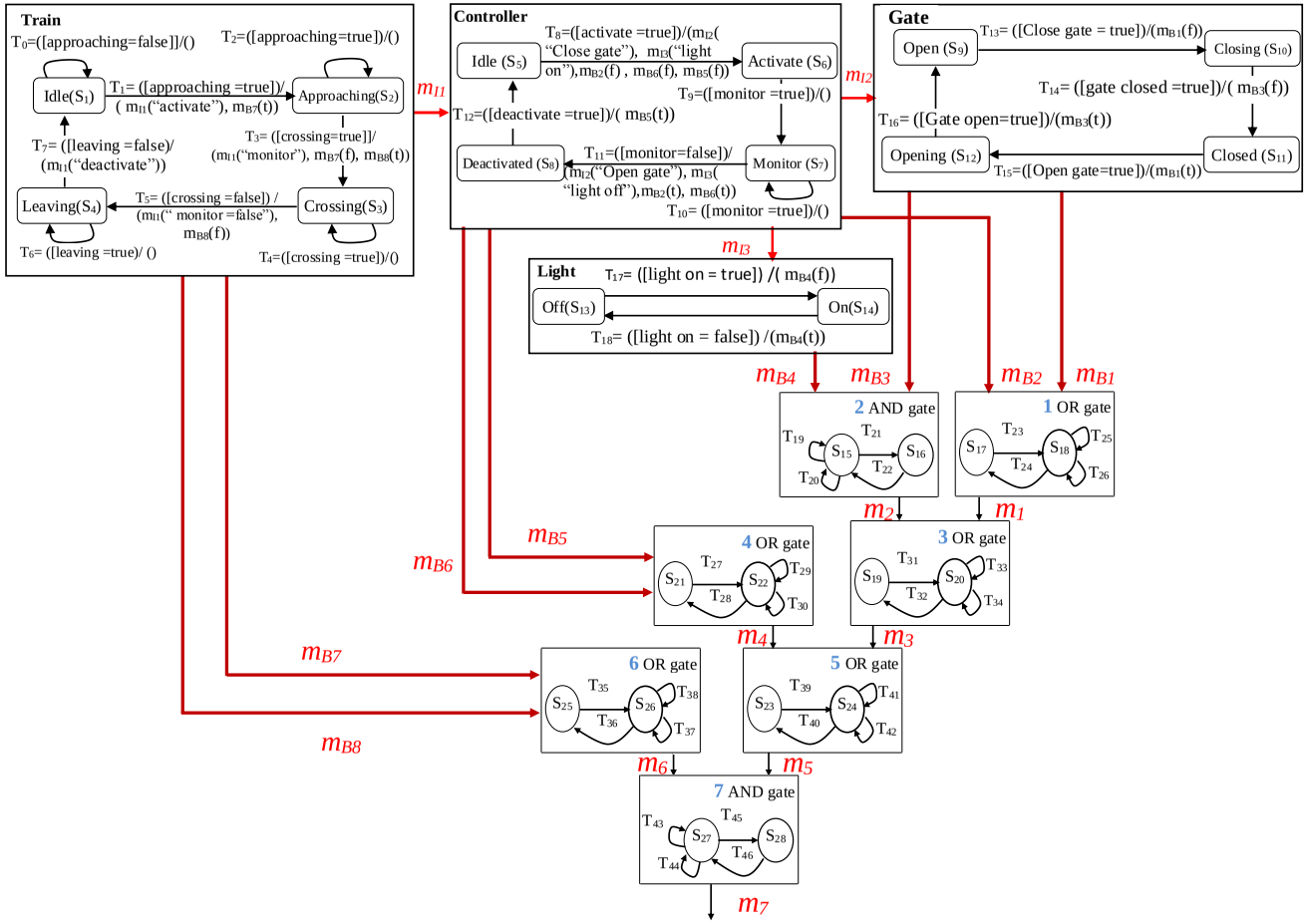

Figure 4.60: The ICEFSM Model of the RCCS

Table 4.25: Failure Types Table After Model Integration Step

Failure ID	Failure Type	Node in FT′	Event ID	Gate ID	Message ID	Path ID
f_1	*Controller Fail*	$BFConrollerFail.BFEventCond$	e_5	G_4	M_{B5}	
f_2	*Warning Light Fail*	$BFWarning.BFEventCond$	e_4	G_2	M_{B4}	
f_3	*Gate Stuck Open*	$BFGateOpen.BFEventCond$	e_3	G_2	M_{B3}	
f_4	*Controller deactivated*	$BFConrollerDeavtivated.BFEventCond$	e_6	G_6	M_{B6}	

Table 4.26: Railroad Crossing System Test Paths

- r_1: Train$[S_1 \xrightarrow{T_2} S_2] \xrightarrow{M_{I1}}$ Controller$[S_6 \xrightarrow{T_9} S_7 \xrightarrow{T_{10}} S_8] \xrightarrow{M_{B5}} [4] \xrightarrow{M_4} [5] \xrightarrow{M_5} [7] \xrightarrow{M_7}$
- r_2: Controller$[S_5 \xrightarrow{T_8} S_6 \xrightarrow{T_9} S_7 \xrightarrow{T_{11}} S_8] \xrightarrow{M_{B6}} [4] \xrightarrow{M_4} [5] \xrightarrow{M_5} [7] \xrightarrow{M_7}$
- r_3: Gate$[S_9 \xrightarrow{T_{13}} S_{10} \xrightarrow{T_{14}} S_{11} \xrightarrow{T_{15}} S_{12}] \xrightarrow{M_{B3}} [2] \xrightarrow{M_3} [5] \xrightarrow{M_5} [7] \xrightarrow{M_7}$
- r_4: Controller$[S_5 \xrightarrow{T_8} S_6 \xrightarrow{T_9} S_7] \xrightarrow{M_{B2}} [1] \xrightarrow{M_1} [3] \xrightarrow{M_3} [5] \xrightarrow{M_5} [7] \xrightarrow{M_7}$
- r_5: Controller$[S_5 \xrightarrow{T_8} S_6 \xrightarrow{T_9} S_7 \xrightarrow{T_{11}} S_8] \xrightarrow{M_{I3}}$ Light$[S_{14} \xrightarrow{T_{18}} S_{13}]$
- r_6: Controller$[S_5] \xrightarrow{M_{I3}}$ Light$[S_{13} \xrightarrow{T_{17}} S_{14}] \xrightarrow{M_{B4}} [2] \xrightarrow{M_2} [3] \xrightarrow{M_3} [5] \xrightarrow{M_5} [7] \xrightarrow{M_7}$
- r_7: Train$[S_1 \xrightarrow{T_1} S_2 \xrightarrow{T_3} S_3 \xrightarrow{T_5} S_4 \xrightarrow{T_6} S_4] \xrightarrow{M_{I1}}$ Controller$[S_8 \xrightarrow{T_{12}} S_5] \xrightarrow{M_{I2}}$ Gate$[S_9 \xrightarrow{T_{13}} S_{10} \xrightarrow{T_{14}} S_{11}] \xrightarrow{M_{B1}} [1] \xrightarrow{M_1} [3] \xrightarrow{M_3} [5] \xrightarrow{M_5} [7] \xrightarrow{M_7}$
- r_8: Train$[S_1 \xrightarrow{T_1} S_2 \xrightarrow{T_3} S_3] \xrightarrow{M_{I1}}$ Controller$[S_7 \xrightarrow{T_{10}} S_7 \xrightarrow{T_{11}} S_8] \xrightarrow{M_{I2}}$ Gate$[S_{11} \xrightarrow{T_{15}} S_{12} \xrightarrow{T_{16}} S_9]$
- r_9: Gate$[S_9 \xrightarrow{T_{13}} S_{10} \xrightarrow{T_{14}} S_{11} \xrightarrow{T_{15}} S_{12}] \xrightarrow{M_{B3}} [2] \xrightarrow{M_2} [3] \xrightarrow{M_3} [5] \xrightarrow{M_5} [7] \xrightarrow{M_7}$
- r_{10}: Controller$[S_5] \xrightarrow{M_{I2}}$ Gate$[S_9 \xrightarrow{T_{13}} S_{10} \xrightarrow{T_{14}} S_{11} \xrightarrow{T_{15}} S_{12}]$
- r_{11}: Train$[S_1 \xrightarrow{T_1} S_2] \xrightarrow{M_{B8}} [6] \xrightarrow{M_6} [7] \xrightarrow{M_7}$
- r_{12}: Train$[S_1] \xrightarrow{M_{I1}}$ Controller$[S_5 \xrightarrow{T_8} S_6 \xrightarrow{T_9} S_7] \xrightarrow{M_{I3}}$ Light$[S_{14} \xrightarrow{T_{18}} S_{13}]$
- r_{13}: Train$[S_1] \xrightarrow{M_{B7}} [6] \xrightarrow{M_6} [7] \xrightarrow{M_7}$
- r_{14}: Train$[S_1 \xrightarrow{T_1} S_2 \xrightarrow{T_3} S_3 \xrightarrow{T_5} S_4 \xrightarrow{T_7} S_1 \xrightarrow{T_1} S_2 \xrightarrow{T_3} S_3 \xrightarrow{T_4} S_3 \xrightarrow{T_5} S_4 \xrightarrow{T_7} S_1]$

4.4.1.6 Test Case Generation from CEFSM Model

If we apply the algorithm in [68] to the graph in Figure 4.60 by imposing edge coverage criteria on the global transitions of the ICEFSM, we will obtain the test paths shown in Table 4.26. By using reachability analysis, we find that these paths are feasible since there are no conflicts between predicates in transitions. Note that we do not need to go into the details of the GCEFSMs and there for we represent each of them as one node. e.x. the GCEFSM 1 that represents the first AND gate is represented as "[1]" and the test path that reaches the root node of the FT, GCEFSM 7, end with a message not a node to indicate that a hazard has occurred. At this step, the last column of the Failure Types Table is updated with the *Test Path* that covers the failure. This is shown in Table 4.27.

4.4.2 Construction of the Applicability Matrix

In addition to test paths r_1-r_{14} in Table 4.26, phase1 produces Table 4.27 that contains failure ID, Failure Type, Failure name in FT', Event ID that carries the failure, Gate ID that takes the failure as an input, and the Message ID of the message that carries the failure. This information is used to map between the failures in the test paths produced in phase1 and the failures that need to be mitigated in phase2. The applicability matrix is build based on the information in this table.

From Table 4.26, we take the test paths through the ICEFSM one by one. We start with r_1. From Table 4.27, we find that r_1 reaches the fault model via M_{B5}. From Table 4.27, we know that M_{B5} is in path r_1 and it carries e_5 which has the failure ID f_1. In the applicability matrix, we assign 1 to the positions indexed (f_1, s_1),(f_1, s_2),(f_1, s_6),(f_1, s_7), and (f_1, s_8). These positions are taken from test path r_1. Next, we look for the behavioral state s_1 which is the starting state of r_1 in other

Table 4.27: Failure Types Table After Test Generation Step

Failure ID	Failure Type	Node in FT′	Event ID	Gate ID	Message ID	Test Path
f_1	Controller Fail	$BFConrollerFail.BFEventCond$	e_5	G_4	M_{B5}	r_1
f_2	Warning Light Fail	$BFWarning.BFEventCond$	e_4	G_2	M_{B4}	r_6
f_3	Gate Stuck Open	$BFGateOpen.BFEventCond$	e_3	G_2	M_{B3}	r_9
f_4	Controller deactivated	$BFConrollerDeavtivated.BFEventCond$	e_6	G_4	M_{B6}	r_2

Table 4.28: Applicability Matrix

F/S	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}
f_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f_2	0	0	0	0	1	1	1	1	0	0	0	0	1	1
f_3	0	0	0	0	0	0	0	0	1	1	1	1	0	0
f_4	0	0	0	0	1	1	1	1	1	1	1	1	1	1

paths that don't contribute to the failure, r_5, r_8, r_{10}, r_{12}, and r_{14}. We assign 1 to the position indexed (f_1, s_5), (f_1, s_6), (f_1, s_7), (f_1, s_{14}), (f_1, s_{13}),(f_1, s_3), (f_1, s_9), (f_1, s_{10}), (f_1, s_{11}), (f_1, s_{12}), and (f_1, s_4) in the applicability matrix shown in table 4.28.
We apply the same steps to the remainder of test paths that contribute to a failure, r_6, r_9, and r_2 that are present in the Failure Types table 4.27.

4.4.3 Phase2: Generate Safety Mitigation Tests

4.4.3.1 Behavioral Model (BM), Test Criteria (BC), and Test Suite (BT)

Figure 4.49 depicts the behavioral model of a Railroad Crossing Control System (RCCS) in Communicating Extended Finite State Machine (CEFSM) format with one train and one track. The model contains 14 states, = $\{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}\}$ where the initial states are s_1,s_5,s_9,s_{13} and the final states are s_1,s_5,s_9,s_{13}. There are 19 transitions T=$\{T_0, T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{15}, T_{16}, T_{17}, T_{18}\}$. Assuming edge coverage is required, the test paths in Table 4.29 fulfill this requirement.

Let r_1-r_{14} be the test paths through ICEFSM in Figure 4.60 obtained from executing phase 1. Let t_1-t_{11} be the paths through the CEFSM shown in Table 4.29.

Table 4.29: Test Paths Through CEFSM Model

Test Paths	sequence
t_1	$s_1 \xrightarrow{T_0} s_1 \xrightarrow{T_1} s_2 \xrightarrow{T_2} s_2 \xrightarrow{T_3} s_3 \xrightarrow{T_4} s_3 \xrightarrow{T_5} s_4 \xrightarrow{T_6} s_4 \xrightarrow{T_7} s_1$
t_2	$s_5 \xrightarrow{T_8} s_6 \xrightarrow{T_9} s_7 \xrightarrow{T_{10}} s_7 \xrightarrow{T_{11}} s_8 \xrightarrow{T_{12}} s_5$
t_3	$s_9 \xrightarrow{T_{13}} s_{10} \xrightarrow{T_{14}} s_{11} \xrightarrow{T_{15}} s_{12} \xrightarrow{T_{16}} s_9$
t_4	$s_{13} \xrightarrow{T_{17}} s_{14} \xrightarrow{T_{18}} s_{13}$
t_5	$s_1 \xrightarrow{T_1} s_2 \xrightarrow{M_{I1}} s_5$
t_6	$s_1 \xrightarrow{T_1} s_2 \xrightarrow{T_3} s_3 \xrightarrow{T_5} s_4 \xrightarrow{M_{I1}} s_7 \xrightarrow{T_{11}} s_8 \xrightarrow{T_{12}} s_5$
t_7	$s_1 \xrightarrow{T_1} s_2 \xrightarrow{T_3} s_3 \xrightarrow{M_{I1}} s_7 \xrightarrow{T_{11}} s_8 \xrightarrow{T_{12}} s_5$
t_8	$s_5 \xrightarrow{T_8} s_6 \xrightarrow{M_{I2}} s_9$
t_9	$s_5 \xrightarrow{T_8} s_6 \xrightarrow{M_{I3}} s_{14} \xrightarrow{T_{18}} s_{13}$
t_{10}	$s_5 \xrightarrow{T_8} s_6 \xrightarrow{T_9} s_7 \xrightarrow{T_{11}} s_8 \xrightarrow{M_{I2}} s_{11} \xrightarrow{T_{15}} s_{12} \xrightarrow{T_{16}} s_9$
t_{11}	$s_5 \xrightarrow{T_8} s_6 \xrightarrow{T_9} s_7 \xrightarrow{T_{11}} s_8 \xrightarrow{M_{I3}} s_{13}$

4.4.3.2 Failure Coverage Criteria (FC)

There are $\sum_{i=1}^{|CT|}$ len(t) positions p to select for failure. Concatenating the tests results in CT = CT= $t_1 \circ t_2 \ldots \circ t_{11} = \{s_1, s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_1, s_5, s_6, s_7, s_7, s_8, s_5, s_9, s_{10}, s_{11}, s_{12}, s_9, s_{13}, s_{14}, s_{13}, s_1, s_2, s_5, s_1, s_2, s_3, s_4, s_7, s_8, s_5, s_1, s_2, s_3, s_7, s_8, s_5, s_5, s_6, s_9, s_5, s_6, s_{14}, s_{13}, s_5, s_6, s_7, s_8, s_{11}, s_{12}, s_9, s_5, s_6, s_7, s_8, s_{13}\}$. There are 58 positions.

We now apply coverage criteria for positions of failure ($1 \leq p \leq 58$) and type of failure ($1 \leq e \leq 4$).

Coverage Criteria 1: all positions, all applicable failures. The required (p,e) combinations are shown in Table 4.30 as "1" entries. 138 tests are required. This is clearly a very large number of tests for a model with only 14 states.

Table 4.30: C1: All Positions, All Applicable Failures

	t_1									t_2						t_3					t_4			t_5			t_6							
F\S	s_1	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_1	s_5	s_6	s_7	s_7	s_8	s_5	s_9	s_{10}	s_{11}	s_{12}	s_9	s_{13}	s_{14}	s_{13}	s_1	s_2	s_5	s_1	s_2	s_3	s_4	s_7	s_8	s_5	
f_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
f_2										1	1	1	1	1	1						1	1	1			1					1	1	1	
f_3																1	1	1	1	1														
f_4										1	1	1	1	1	1	1	1	1	1	1	1	1	1			1					1	1	1	

	t_7						t_8			t_9				t_{10}							t_{11}					
F\S	s_1	s_2	s_3	s_7	s_8	s_5	s_5	s_6	s_9	s_5	s_6	s_{14}	s_{13}	s_5	s_6	s_7	s_8	s_{11}	s_{12}	s_9	s_5	s_6	s_7	s_8	s_{13}	
f_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
f_2				1	1	1	1	1		1	1	1	1	1	1	1	1				1	1	1	1	1	
f_3									1									1	1	1						
f_4				1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

Table 4.31: C2: All Unique Nodes, All Applicable Failures

	t_1									t_2						t_3					t_4			t_5			t_6							
F\S	s_1	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_1	s_5	s_6	s_7	s_7	s_8	s_5	s_9	s_{10}	s_{11}	s_{12}	s_9	s_{13}	s_{14}	s_{13}	s_1	s_2	s_5	s_1	s_2	s_3	s_4	s_7	s_8	s_5	
f_1	1		1		1		1			1	1	1		1		1	1	1	1		1	1											
f_1										1	1	1		1							1	1											
f_3																1	1	1	1														
f_4										1	1	1		1		1	1	1	1		1	1											

	t_7						t_8			t_9				t_{10}							t_{11}					
F\S	s_1	s_2	s_3	s_7	s_8	s_5	s_5	s_6	s_9	s_5	s_6	s_{14}	s_{13}	s_5	s_6	s_7	s_8	s_{11}	s_{12}	s_9	s_5	s_6	s_7	s_8	s_{13}	
f_1																										
f_2																										
f_3																										
f_4																										

Table 4.32: C3: All Tests, All Unique Nodes, All Applicable Failures

	t_1									t_2						t_3					t_4			t_5			t_6							
F\S	s_1	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_1	s_5	s_6	s_7	s_7	s_8	s_5	s_9	s_{10}	s_{11}	s_{12}	s_9	s_{13}	s_{14}	s_{13}	s_1	s_2	s_5	s_1	s_2	s_3	s_4	s_7	s_8	s_5	
f_1	1						1			1							1		1	1		1			1				1					
f_2									1													1												
f_3																	1		1	1														
f_4									1								1		1	1			1											

	t_7						t_8			t_9				t_{10}							t_{11}					
F\S	s_1	s_2	s_3	s_7	s_8	s_5	s_5	s_6	s_9	s_5	s_6	s_{14}	s_{13}	s_5	s_6	s_7	s_8	s_{11}	s_{12}	s_9	s_5	s_6	s_7	s_8	s_{13}	
f_1				1				1						1					1						1	
f_2			1					1					1											1		
f_3																		1								
f_4			1					1					1					1						1		

Table 4.33: C4: All Tests, All Unique Nodes, Some Applicable Failures

	t_1									t_2						t_3					t_4			t_5			t_6							
F\S	s_1	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_1	s_5	s_6	s_7	s_7	s_8	s_5	s_9	s_{10}	s_{11}	s_{12}	s_9	s_{13}	s_{14}	s_{13}	s_1	s_2	s_5	s_1	s_2	s_3	s_4	s_7	s_8	s_5	
f_1	1						1												1			1			1				1					
f_2										1																								
f_3																1	1																	
f_4																																		

	t_7						t_8			t_9				t_{10}							t_{11}					
F\S	s_1	s_2	s_3	s_7	s_8	s_5	s_5	s_6	s_9	s_5	s_6	s_{14}	s_{13}	s_5	s_6	s_7	s_8	s_{11}	s_{12}	s_9	s_5	s_6	s_7	s_8	s_{13}	
f_1																		1								
f_2			1										1											1		
f_3																										
f_4								1																		

Coverage Criteria 2: all unique nodes, all applicable failures. The required position-failure pairs (p,e) are shown in Table 4.31 as "1" entries. This required 34 tests. Note that behavioral test paths $t_5 - t_{11}$ are not used. This criterion leads to not testing failure recovery in all tests. A stronger test criterion is to require covering each test as well. The (p,e) pairs are also stated in the second column in Table 4.35.

Coverage Criteria 3: all test, all unique nodes, all applicable failures. Here we simply require that when unique nodes need to be covered they are selected from tests that have not been covered. Table 4.32 shows the set of position-failure pairs (p, e) that fulfills this criteria. As with criteria 2, this required 34 pairs. The (p, e) pairs are stated in the second column in Table 4.36.

Coverage Criteria 4: all tests, all unique nodes, some failures. The required (p,e) pairs are shown in Table 4.33 as "1" entries. This requires 14 tests. The (p,e) pairs are also stated in the second column in Table 4.37 to indicate the associated safety mitigation tests.

4.4.3.3 Mitigation Requirements, Mitigation Models, and Safety Mitigation Tests

The mitigation requirements are summarized in Table 4.34. Table 4.34 specifies the corresponding mitigation models and associated weaving rules. Figures 4.61-4.63 show the mitigation models for failures $f_2 - f_4$. Note that f_1 does not need a model, since it is an implicit fix that does not use additional test inputs (category 4 under weaving rules). Again, assuming edge coverage, the mitigation tests listed in figures 4.61- 4.63 fulfill this coverage. Note also, that only failure f_4 has more than one mitigation test path.

Table 4.34: Mitigation Requirement

MM	Explanation	Mitigation Model	WR
MM_1	compensate; switch to backup sensor (internal action); send alarm	none(internal compensate);	4 in 4.3.4.2
MM_2	fix and stop; close gate; send alarm; stop	*cf.* Figure 4.61	1b in 4.3.4.2
MM_3	fix and proceed; turn warning light on; send alarm	*cf.* Figure 4.62	1a in 4.3.4.2
MM_4	Try other alternatives: compensate; switch to back up and close gates or turn warning light on; send maintenance request	*cf.* Figure 4.63	1a in 4.3.4.2

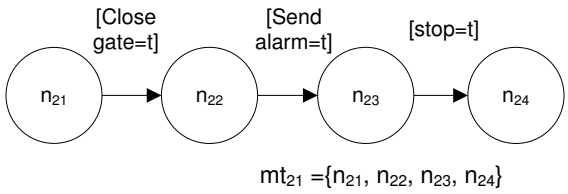

Figure 4.61: Fix and Stop: Mitigation Model MM_2

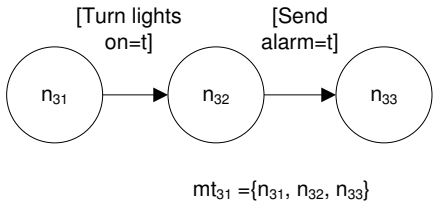

Figure 4.62: Fix and Proceed: Mitigation Model MM_3

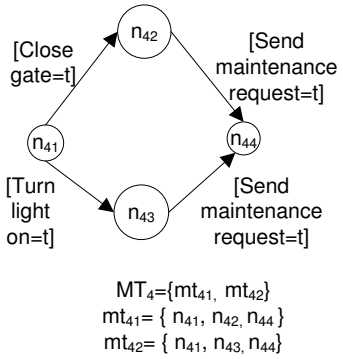

Figure 4.63: Compensate: Mitigation Model MM_4

Construct Safety Mitigation Tests: Due to the large number of tests for C1, we will only show tests for criteria C2-C4. Table 4.35 indicates tests for the 44 position-failure pairs that fulfill coverage criteria 2. Note that because f_4 has two mitigation paths required, these are two test paths for each position failure pair $(i, 4)$ $(i \in 10-12, 14, 16-19, 21-22)$.

Table 4.35: Safety Mitigation Tests for Criteria 2

SMT	Covers	Explanation	BT used
smt_1	(1,1)	t_1 (no MT added to it)	t_1
smt_2	(3,1)	t_1 (no MT added to it)	t_1
smt_3	(5,1)	t_1 (no MT added to it)	t_1
smt_4	(7,1)	t_1 (no MT added to it)	t_1
Continued on next page			

Continued from previous page

SMT	Covers	Explanation	BT used
smt_5	(10,1)	t_2 (no MT added to it)	t_2
smt_6	(11,1)	t_2 (no MT added to it)	t_2
smt_7	(12,1)	t_2 (no MT added to it)	t_2
smt_8	(14,1)	t_2 (no MT added to it)	t_2
smt_9	(16,1)	t_3 (no MT added to it)	t_3
smt_{10}	(17,1)	t_3 (no MT added to it)	t_3
smt_{11}	(18,1)	t_3 (no MT added to it)	t_3
smt_{12}	(19,1)	t_3 (no MT added to it)	t_3
smt_{13}	(21,1)	t_4 (no MT added to it)	t_4
smt_{14}	(22,1)	t_4 (no MT added to it)	t_4
smt_{15}	(10,2)	$s_5, n_{21}, n_{22}, n_{23}, n_{24}$	t_2
smt_{16}	(11,2)	$s_5, s_6, n_{21}, n_{22}, n_{23}, n_{24}$	t_2
smt_{17}	(12,2)	$s_5, s_6, s_7, n_{21}, n_{22}, n_{23}, n_{24}$	t_2

Continued on next page

Continued from previous page

SMT	Covers	Explanation	BT used
smt_{18}	(14,2)	$s_5, s_6, s_7, s_7, s_8, n_{21}, n_{22}, n_{23}, n_{24}$	t_2
smt_{19}	(21,2)	$s_{13}, n_{21}, n_{22}, n_{23}, n_{24}$	t_4
smt_{20}	(22,2)	$s_{13}, s_{14}, n_{21}, n_{22}, n_{23}, n_{24}$	t_4
smt_{21}	(16,3)	$s_9, n_{31}, n_{32}, n_{33}, s_9, s_{10}, s_{11}, s_{12}, s_9$	t_3
smt_{22}	(17,3)	$s_9, s_{10}, n_{31}, n_{32}, n_{33}, s_{10}, s_{11}, s_{12}, s_9$	t_3
smt_{23}	(18,3)	$s_9, s_{10}, s_{11}, n_{31}, n_{32}, n_{33}, s_{11}, s_{12}, s_9$	t_3
smt_{24}	(19,3)	$s_9, s_{10}, s_{11}, s_{12}, n_{31}, n_{32}, n_{33}, s_{12}, s_9$	t_3
smt_{25}	(10,4)	$s_5, n_{41}, n_{42}, n_{44}, s_5, s_6, s_7, s_7, s_8, s_5$	t_2
smt_{26}	(10,4)	$s_5, n_{41}, n_{43}, n_{44}, s_5, s_6, s_7, s_7, s_8, s_5$	t_2
smt_{27}	(11,4)	$s_5, s_6, n_{41}, n_{42}, n_{44}, s_6, s_7, s_7, s_8, s_5$	t_2
smt_{28}	(11,4)	$s_5, s_6, n_{41}, n_{43}, n_{44}, s_6, s_7, s_7, s_8, s_5$	t_2
smt_{29}	(12,4)	$s_5, s_6, s_7, n_{41}, n_{42}, n_{44}, s_7, s_7, s_8, s_5$	t_2
smt_{30}	(12,4)	$s_5, s_6, s_7, n_{41}, n_{43}, n_{44}, s_7, s_7, s_8, s_5$	t_2

Continued on next page

Continued from previous page

SMT	Covers	Explanation	BT used
smt_{31}	(14,4)	$s_5, s_6, s_7, s_7, s_8, n_{41}, n_{42}, n_{44}, s_8, s_5$	t_2
smt_{32}	(14,4)	$s_5, s_6, s_7, s_7, s_8, n_{41}, n_{43}, n_{44}, s_8, s_5$	t_2
smt_{33}	(16,4)	$s_9, n_{41}, n_{42}, n_{44}, s_9, s_{10}, s_{11}, s_{12}, s_9$	t_3
smt_{34}	(16,4)	$s_9, n_{41}, n_{43}, n_{44}, s_9, s_{10}, s_{11}, s_{12}, s_9$	t_3
smt_{35}	(17,4)	$s_9, s_{10}, n_{41}, n_{42}, n_{44}, s_{10}, s_{11}, s_{12}, s_9$	t_3
smt_{36}	(17,4)	$s_9, s_{10}, n_{41}, n_{43}, n_{44}, s_{10}, s_{11}, s_{12}, s_9$	t_3
smt_{37}	(18,4)	$s_9, s_{10}s_{11}, n_{41}, n_{42}, n_{44}, s_{11}, s_{12}, s_9$	t_3
smt_{38}	(18,4)	$s_9, s_{10}, s_{11}, n_{41}, n_{43}, n_{44}, s_{11}, s_{12}, s_9$	t_3
smt_{39}	(19,4)	$s_9, s_{10}, s_{11}, s_{12}, n_{41}, n_{42}, n_{44}, s_{12}, s_9$	t_3
smt_{40}	(19,4)	$s_9, s_{10}, s_{11}, s_{12}, n_{41}, n_{43}, n_{44}, s_{12}, s_9$	t_3
smt_{41}	(21,4)	$s_{13}, n_{41}, n_{42}, n_{44}, s_{13}, s_{14}, s_{13}$	t_4
smt_{42}	(21,4)	$s_{13}, n_{41}, n_{43}, n_{44}, s_{13}, s_{14}, s_{13}$	t_4
smt_{43}	(22,4)	$s_{13}, s_{14}, n_{41}, n_{42}, n_{44}, s_{14}, s_{13}$	t_4

Continued on next page

			Continued from previous page
SMT	Covers	Explanation	BT used
smt_{44}	(22,4)	$s_{13}, s_{14}, n_{41}, n_{43}, n_{44}, s_{14}, s_{13}$	t_4

Table 4.36 lists safety mitigation tests for the 44 position-failure pairs that fulfill coverage criteria 3.

Table 4.36: Safety Mitigation Tests for Criteria 3

SMT	Covers	Explanation	BT used
smt_1	(1,1)	t_1 (no MT added to it)	t_1
smt_2	(7,1)	t_1 (no MT added to it)	t_1
smt_3	(10,1)	t_2 (no MT added to it)	t_2
smt_4	(17,1)	t_3 (no MT added to it)	t_3
smt_5	(19,1)	t_3 (no MT added to it)	t_3
smt_6	(20,1)	t_3 (no MT added to it)	t_3
smt_7	(22,1)	t_4 (no MT added to it)	t_4
smt_8	(25,1)	t_5 (no MT added to it)	t_5
			Continued on next page

Continued from previous page

SMT	Covers	Explanation	BT used
smt_9	(29,1)	t_6 (no MT added to it)	t_6
smt_{10}	(37,1)	t_7 (no MT added to it)	t_7
smt_{11}	(41,1)	t_8 (no MT added to it)	t_8
smt_{12}	(46,1)	t_9 (no MT added to it)	t_9
smt_{13}	(51,1)	t_{10} (no MT added to it)	t_{10}
smt_{14}	(57,1)	t_{11} (no MT added to it)	t_{11}
smt_{15}	(10,2)	$s_5, n_{21}, n_{22}, n_{23}, n_{24}$	t_2
smt_{16}	(22,2)	$s_{13}, s_{14}, n_{21}, n_{22}, n_{23}, n_{24}$	t_4
smt_{17}	(37,2)	$s_1, s_2, s_3, s_7, n_{21}, n_{22}, n_{23}, n_{24}$	t_7
smt_{18}	(41,2)	$s_5, s_6, n_{21}, n_{22}, n_{23}, n_{24}$	t_8
smt_{19}	(46,2)	$s_5, s_6, s_{14}, s_{13}, n_{21}, n_{22}, n_{23}, n_{24}$	t_9
smt_{20}	(57,2)	$s_5, s_6, s_7, s_8, n_{21}, n_{22}, n_{23}, n_{24}$	t_{11}
smt_{21}	(17,3)	$s_9, s_{10}, n_{31}, n_{32}, n_{33}, s_{10}, s_{11}, s_{12}, s_9$	t_3

Continued on next page

Continued from previous page

SMT	Covers	Explanation	BT used
smt_{22}	(19,3)	$s_9, s_{10}, s_{11}, s_{12}, n_{31}, n_{32}, n_{33}, s_{12}, s_9$	t_3
smt_{23}	(20,3)	$s_9, s_{10}, s_{11}, s_{12}, s_9, n_{31}, n_{32}, n_{33}, s_9$	t_3
smt_{24}	(51,3)	$s_5, s_6, s_7, s_8, s_{11}, n_{31}, n_{32}, n_{33}, s_{11}, s_{12}, s_9$	t_{10}
smt_{25}	(10,4)	$s_5, n_{41}, n_{42}, n_{44}, s_5, s_6, s_7, s_7, s_8, s_5,$	t_2
smt_{26}	(10,4)	$s_5, n_{41}, n_{43}, n_{44}, s_5, s_6, s_7, s_7, s_8, s_5,$	t_2
smt_{27}	(17,4)	$s_9, s_{10}, n_{41}, n_{42}, n_{44}, s_{10}, s_{11}, s_{12}, s_9$	t_3
smt_{28}	(17,4)	$s_9, s_{10}, n_{41}, n_{43}, n_{44}, s_{10}, s_{11}, s_{12}, s_9$	t_3
smt_{29}	(19,4)	$s_9, s_{10}, s_{11}, s_{12}, n_{41}, n_{42}, n_{44}, s_{12}, s_9$	t_3
smt_{30}	(19,4)	$s_9, s_{10}, s_{11}, s_{12}, n_{41}, n_{43}, n_{44}, s_{12}, s_9$	t_3
smt_{31}	(20,4)	$s_9, s_{10}, s_{11}, s_{12}, s_9, n_{41}, n_{42}, n_{44}, s_9$	t_3
smt_{32}	(20,4)	$s_9, s_{10}, s_{11}, s_{12}, s_9, n_{41}, n_{43}, n_{44}, s_9$	t_3
smt_{33}	(22,4)	$s_{13}, n_{41}, n_{42}, n_{44}, s_{14}, s_{13}$	t_4
smt_{34}	(22,4)	$s_{13}, n_{41}, n_{43}, n_{44}, s_{14}, s_{13}$	t_4

Continued on next page

Continued from previous page			
SMT	Covers	Explanation	BT used
smt_{35}	(37,4)	$s_1, s_2, s_3, s_7, n_{41}, n_{42}, n_{44}, s_7, s_8, s_5$	t_7
smt_{36}	(37,4)	$s_1, s_2, s_3, s_7, n_{41}, n_{43}, n_{44}, s_7, s_8, s_5$	t_7
smt_{37}	(41,4)	$s_5, s_6, n_{41}, n_{42}, n_{44}, s_6, s_9$	t_8
smt_{38}	(41,4)	$s_5, s_6, n_{41}, n_{43}, n_{44}, s_6, s_9$	t_8
smt_{39}	(46,4)	$s_5, s_6, s_{14}, s_{13}, n_{41}, n_{42}, n_{44}, s_{13}$	t_9
smt_{40}	(46,4)	$s_5, s_6, s_{14}, s_{13}, n_{41}, n_{43}, n_{44}, s_{13}$	t_9
smt_{41}	(51,4)	$s_5, s_6, s_7, s_8, s_{11}, n_{41}, n_{42}, n_{44}, s_{11}, s_{12}, s_9$	t_{10}
smt_{42}	(51,4)	$s_5, s_6, s_7, s_8, s_{11}, n_{41}, n_{43}, n_{44}, s_{11}, s_{12}, s_9$	t_{10}
smt_{43}	(57,4)	$s_5, s_6, s_7, s_8, n_{41}, n_{42}, n_{44}, s_8, s_{13}$	t_{11}
smt_{44}	(57,4)	$s_5, s_6, s_7, s_8, n_{41}, n_{43}, n_{44}, s_8, s_{13}$	t_{11}

Table 4.37 indicates tests for the 15 position-failure pairs that fulfill coverage criteria 4.

Table 4.37: Safety Mitigation Tests for Criteria 4

SMT	Covers	Explanation	BT used
smt_1	(1,1)	t_1 (no MT added to it)	t_1
smt_2	(7,1)	t_1 (no MT added to it)	t_1
smt_3	(19,1)	t_3 (no MT added to it)	t_3
smt_4	(22,1)	t_4 (no MT added to it)	t_4
smt_5	(25,1)	t_5 (no MT added to it)	t_5
smt_6	(29,1)	t_6 (no MT added to it)	t_6
smt_7	(51,1)	t_{10} (no MT added to it)	t_{10}
smt_8	(10,2)	$s_5, n_{21}, n_{22}, n_{23}, n_{24}$	t_2
smt_9	(37,2)	$s_1, s_2, s_3, s_7, n_{21}, n_{22}, n_{23}, n_{24}$	t_7
smt_{10}	(46,2)	$s_5, s_6, s_{14}, s_{13}, n_{21}, n_{22}, n_{23}, n_{24}$	t_9
smt_{11}	(57,2)	$s_5, s_6, s_7, s_8, n_{21}, n_{22}, n_{23}, n_{24}$	t_{11}
smt_{12}	(16,3)	$s_9, n_{31}, n_{32}, n_{33}, s_9, s_{10}, s_{11}, s_{12}, s_9$	t_3
smt_{13}	(17,3)	$s_9, s_{10}, n_{31}, n_{32}, n_{33}, s_{10}, s_{11}, s_{12}, s_9$	t_3

Continued on next page

Continued from previous page

SMT	Covers	Explanation	BT used
smt_{14}	(41,4)	$s_5, s_6, n_{41}, n_{42}, n_{44}, s_5, s_6, s_9$	t_8
smt_{15}	(41,4)	$s_5, s_6, n_{41}, n_{43}, n_{44}, s_5, s_6, s_9$	t_8

Continued on next page

Criteria 1 would need $|I| \times |F|$ minus the zeros entries (not applicable) in Table 4.30. This requires $58 \times 4 - 98 = 134$ tests which is clearly not desirable. Criteria 2 and 3 have $34 + 10 = 44$ tests and Criteria 4 have 15 tests. When deciding which criteria to use, some other factors might need to be considered:

1. The likelihood that dependencies (that are not transparent in black-box testing) between behavioral states and failure types could expose mitigation defects in some states, but not others (this would require criteria 2 or 3).
2. The likelihood that specific execution history until the point the failure is applied impacts the probability of uncovering a mitigation defect (this would require criteria 3).
3. The cost of testing and the risk of missing a mitigation defect. This could result in applying C1 to some failure types, but not to all. In other words, the testing criteria are flexible enough to consider them each individually.

Either of those renders criteria 4 inadequate. It would probably also be wise to consider the severity of a failure not being properly mitigated and using this knowledge to prioritize tests. However, with FT as a model used in qualitative,

rather than quantitative analysis, this information is not included in a FT. One would have to switch to a more quantitative failure "model" such as the information available in FMECA [119].

End-to-End methodology consists of two phases. In phase 1, we integrated the behavioral and fault models to produce test cases from the integrated model. This phase starts with the compatibility transformation step, in which we transformed the fault trees to become compatible with the behavioral model, followed by model transformation step. In this step, the fault tree was transformed into GCEFSM according to the transformation rules. In the model integration step where we integrate the GCEFSM with the behavioral model according to the integration rules to produce the ICEFSM model which is then used to produce 14 test paths Table 4.26. This phase also produced the failure type table (Table 4.27) which was used by the applicability matrix construction procedure (Figure 4.47) along with the test paths in (Table 4.26) to construct the applicability matrix (Table 4.28). This table contains 4 failure types with the information that shows which test path the failure is feasible in. The applicability matrix is taken as an input to phase 2.

In phase 2, our aim is to provide an MBT approach to test proper mitigation of safety failures in SCSs. We used the applicability matrix to build test criteria from which we select the position and the type of the failure. The number of the behavior test paths (without mitigation tests) produced from the behavioral model is 11 paths and the total number of mitigation tests (MT) that we have is 4. Some failures require more than one mitigation test paths (2 test paths), such as ($f4$) which caused the number of safety mitigation tests (SMTs) to increase. From these path, we generated SMTs for each criteria. The safety mitigation tests (SMTs) are 44 for criteria 2 and criteria 3. For criteria 4, we have 15 SMTs. The increase of the number of the test paths with mitigation tests (SMTs) can be considered reasonable.

Chapter 5

Other Uses for Integrated Model

5.1 Additional Analysis Capabilities through Construction and Analysis of Distributed Processes (CADP)

5.1.1 Construction and Analysis of Distributed Processes (CADP)

CADP [142], formerly known as "CAESAR/ALDEBARAN Development Package", is a toolbox for communication systems engineering. CADP's development started in 1986 by the VASY team of INRIA and the Verimag laboratory with contributions of the PAMPA team of the Institute for Research in IT and Random Systems (IRISA) and the Formal Methods and Tools (FMT) group at the University of Twente. CADP is a tool for verifying asynchronous concurrent systems. It consists of 45 tools that offer a set of functionalities that cover the design cycle of asynchronous systems such as specification, interactive simulation, rapid prototyping, verification, testing, and performance evaluation [50].

CADP can be seen as a rich set of powerful, interoperating software components. All tools are integrated for interactive use via a graphical user-interface (i.e. Eucalyptus) and for batch use via a user-friendly scripting language (SVL). CADP can manage as large as 10^{10} explicit states and much larger state spaces can be handled by employing compositional verification techniques on individual processes [50]. Because the textual file format that was used in the early 90s by most verification tools is adequate for small graphs, CADP was equipped in 1994 with binary-coded graphs (BCG), a portable file format for sorting labeled transition systems (LTSs). BCG is capable of handling large state spaces (up to 10^8 states and transitions initially, this limit was raised to 10^{13} in CADP 2011 for 64-bit machines) [50].

Garavel *et al.* [51] Explored the distributed state spaces of a large-scale grid involving several clusters by the distributed verification tools recently added to the CADP toolbox. These experiments were intended to push the PBG machinery to its limits to study how this influences performance and scalability. They found that CADP can handle about 289,130,000 states and 542,000,000 transitions for a Dijkstra protocol of 4 processes. Compositional verification techniques offered by CADP is applied by Garavel *et. al.* [49] a graph of 155,377,200 states and 371,146,000 transitions.

5.1.2 Process

To be able to use CADP to generate test cases, we can enter the ICEFSM description directly into CADP, converting it to LOTOS or LOTOS NT. This, however, does not automate the integration process. Hence, we are in the process of implementing a front-end tool[1] that converts the fault tree into GCEFSM in the

[1]This front-end tool is a collaboration work between the University of Denver and the University of North Dakota. It is being implemented by Amro Hassan and Mitchell Wright.

LOTOS format. We are also implementing a tool that takes a CEFSM behavioral model and converts it into LOTOS format. After the conversion is done, we will be integrating the models as we described in section 3.1.7. The result of the front-end component which is an integrated CEFSM (ICEFSM) written in LOTOS is taken by CADP and transformed into an Labeled Transition System (LTS).

5.1.2.1 Modeling with Labeled Transition Systems (LTSs)

LTSs [80] have been used to precisely represent the semantics of behavioral specifications. LTSs are used to reason about processes, such as specification, implementations, and tests. In general, an LTS provides a global monolithic description of the set of all possible behaviors of the system. It differentiates between internal and external actions. LTSs are represented by graphs of states and edges. The states represent configurations of systems, and the edges represent the moves between these configurations on the occurrence of actions. A labeled transition system (LTS) is a tuple M = (Q, A, T, q_0) where

- Q $\neq$ 0 is a set of states,
- A is a set of actions (machine alphabet),
- T $\subseteq$ Q $\times$ A $\times$ Q is a transition relation between two states q, $q' \in$ Q, connected by an action (a label) a $\in$ A, denoted (q, a, q') $\in$ T or q $\xrightarrow{a}$ $q' \in$ T, and
- $q_0 \in$ Q is the initial state.

The elements $a \in A$ are the actions of the LTS. They are also referred to as labels.

However, except for the most trivial systems, a visual representation by means of a tree or a graph is not feasible. Realistic systems would normally have billions of LTS states, therefore drawing them is not an option [137]. Figure 5.2 shows the LTS representation of the 2-state-3-transition EFSM model shown in Figure 5.1.

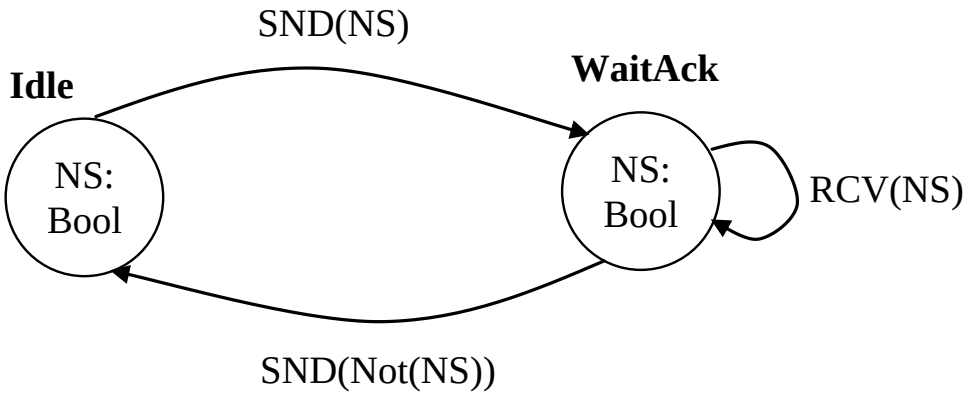

Figure 5.1: BitAlt Sender Protocol EFSM

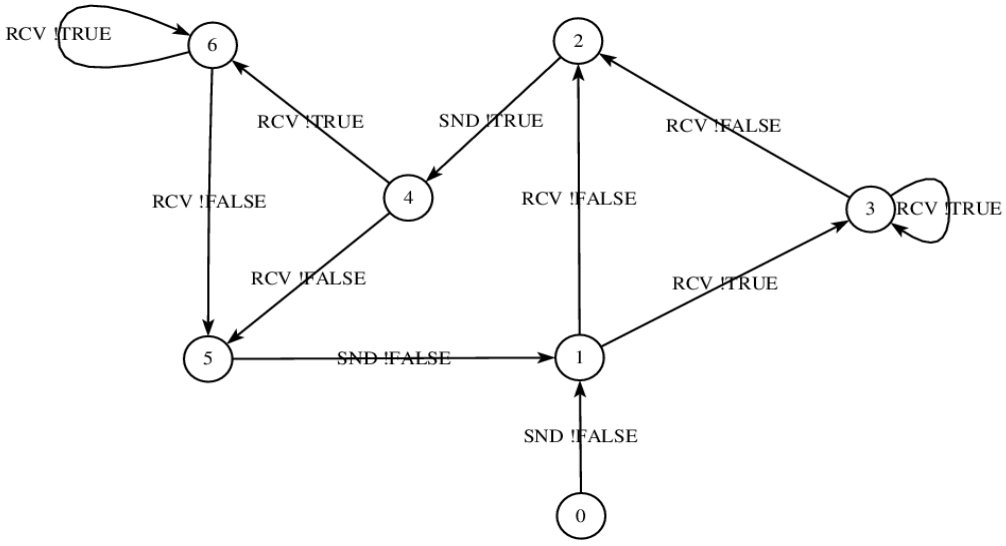

Figure 5.2: BitAlt Sender Protocol LTS (CADP Produced Graph)

5.1.3 Deadlock

A deadlock for (parallel compositions of) LTSs, or any other systems on which we can perform reachability analysis is a global state v in the reachability graph such that there is no transition going out of the state v. Deadlocks are a common problem in distributed systems. In a communicating processes or a message passing system, deadlocks might occur due to processes indefinitely waiting for messages from one another. Since our integrated model ICEFM is a collection of communicating processes, it is necessary to evaluate it for deadlock in order to know that the system is communicating as it should and no process is waiting for any communicating message that will never arrive. The deadlock evaluation is done during the compilation of the model from the LOTOS to LTS and it can be separately on the LTS model.

In addition to evaluating the whole mode for deadlock, we can use deadlock to evaluate safety. This property can be used to analyze safety in our integrated model ICEFSM, i.e. the deadlock state is the state that can not lead to any hazardous state in the integrated model. In other words, if deadlock occurs in any state at the fault side of the model, this means that the hazard state cannot be reached from this state and that indicates that the events lead to this state will not cause a hazard. Indeed, CADP offers checking this property by exploring a random sequence in the model until a non-Markovian transition or a deadlock state is found, or it reaches the maximum values specified by the user, or the simulation is halted by the user. We used CADP to find whether there is a deadlock state in the example in Figure 5.2, the result was no deadlock states were found (*cf* Figure 5.3).

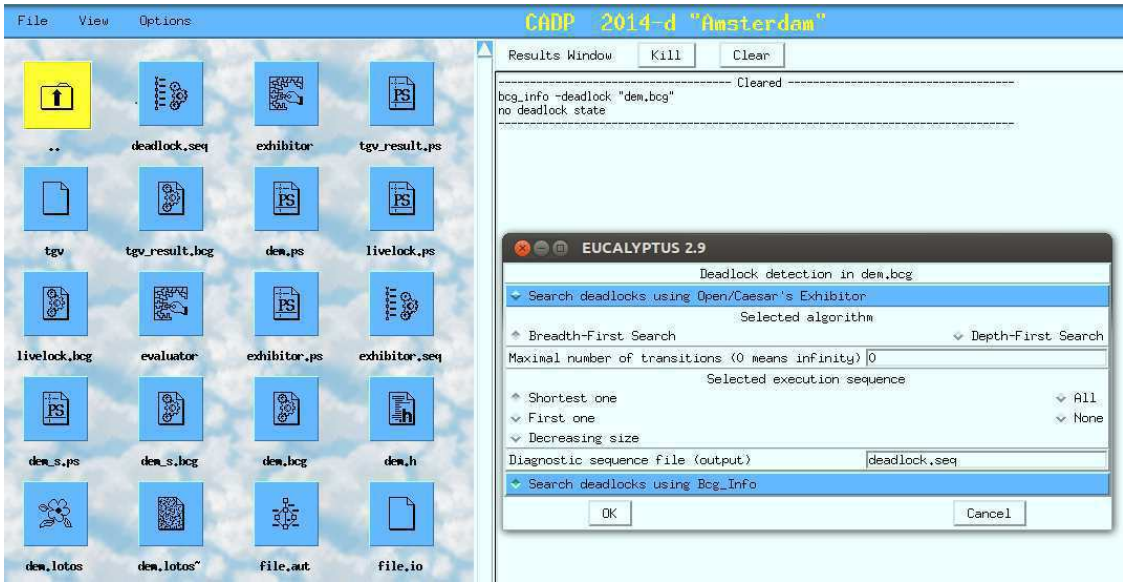

Figure 5.3: CADP Deadlock Screen

5.1.4 Livelock

A livelock property (also called divergence) is when one or more processes enter an infinite cycle where no progress occurs. It is similar to deadlock, except that the involved processes exchange messages and change states with regard to one another while no "useful work" is being done. As for an LTS, a livelock exists in a state s of an LTS L, if s situated in a loop of internal actions, although it possibly has other outgoing transitions. Brzeziński *et al.* define livelock property for communicating processes as follows: Let C_1, C_2, ..., C_n be directed cycles in entities ent_1, ent_2, ..., ent_2 of protocol Pr, respectively. The tuple (C_1, C_2, ..., C_n) is called livelock in protocol Pr, iff there exisits a sequence (s_1, s_2, ..., s_r) of reachable states of protocls Pr such that the following conditions hold:

- For i =1,..., r+1, state s_{i-1} follows s_i over an edge e_1 in ent_1, or ent_2,..., or ent_n. Also states s_1 follows s_r over an edge e_r in ent_1, or ent_2,...., or ent_n.
- The set of edges {e_1, e_2, ..., e_r} constitutes of cycles C_1, C_2, ..., C_n.

- the sequence (s_1,..., s_r) is unacceptable.

A livelock corresponds to a cycle of internal transitions reachable from s using only internal transitions. It can be considered as a loop of internal actions. Similar to the deadlocks, livelocks are often considered a faulty behavior. CADP analyzes livelocks consisting of loops due to internal (unobservable) actions causing the system to loop forever. This implies that the external observer will not see any progress in the system. This property can be used to verify whether a GCEFSM (a gate CEFSM in the fault tree part of the model) receives a message that triggers an output message ("gate occurred" or "gate not occurred") or not. If that GCEFSM has not sent a message, this implies that either it has not received any message or it has received messages in an order that does not trigger the gate to occur which implies that the input events to that gate may not contribute in a failure.

Figure 5.4 shows the livelock screen produced by CADP from the example in Figure 5.2 .

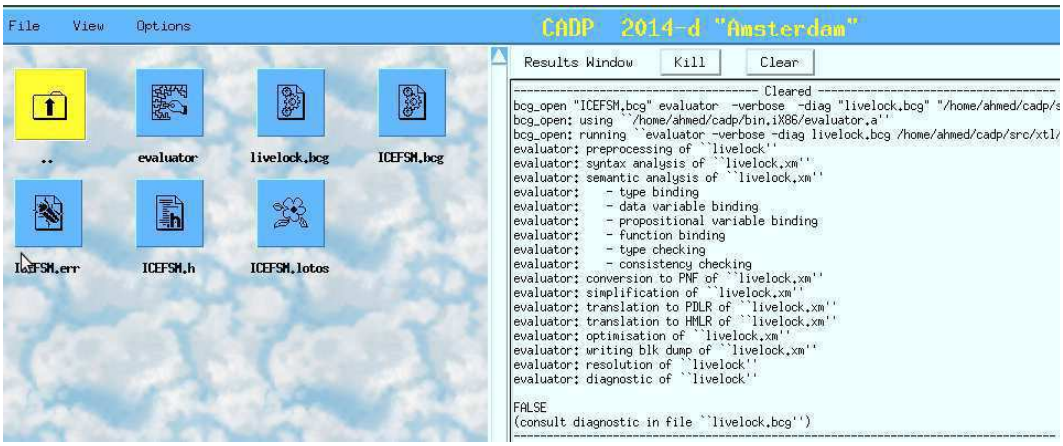

Figure 5.4: CADP Livelock Screen

5.1.5 Test Generation with Verification (TGV)

The test generator TGV is part of CADP. It is available as command line and as part of CADP's graphical environment *Eucalyptus*. TGV uses the simulation API provided by the CAESAR compiler of the CADP toolbox. It produces test cases from the deterministic input-output labeled transition system (IOLTS) behavioral model. Since the produced IOLTS model is normally big for a reasonable system [137], we need to describe what can be called a test selection directive. Test selection directives may be in the form of random test selection, selection based on some kind of coverage criteria, selection described by test purposes, or mixture of these. The test selection directives are a description of a targeted behavior that one needs to test.

IOLTS is a tuple $M = (Q,\ A,\ T,\ q_0)$. The difference to LTSs is the distinction of the actions. IOLTS divides the actions into three subsets, input (visible actions) A_I , output (visible actions) A_O , internal (invisible actions) I.

TGV generates abstract test cases that describe the behavior of the system in terms of input/output interaction between the tester and the implementation under test (IUT) and the verdicts associated with these behaviors [74]. The produced test cases are in the form of a graph such as Tree and Tabular Combined Notation (TTCN) or in one of the graph formats (.aut and .bcg) of the CADP. The algorithm of TGV can be described as:

1. TGV takes a specification (S) and a test purpose (TP) as inputs.
2. TGV performs a synchronous product (SP) between S and TP, marking the S's behavior accepted or refused by TP. The *synchronous product* $S \times TP$ *is an IOLTS* , equipped with two disjoint sets of states $Accept^{SP}$ and $Refuse^{SP}$, and defined as follows:

Given $M^S = (Q^S, A^S, T^S, q_0^S)$, $M^{TP} = (Q^{TP}, A^{TP}, T^{TP}, q_0^{TP})$, their synchronous product is an IOLTS $M^{S\times TP} = (Q^{S\times TP}, A^{S\times TP}, T^{S\times TP}, q_0^{S\times TP})$ such that :[24]

- $q_0^{S\times TP} = (q_0^S, q_0^{TP})$,
- $Q^{S\times TP} = Q^S \times Q^{TP}$,
- $A^{S\times TP} = A^S \bigcap A^{TP}$,
- $(q^S, q^{TP}) \xrightarrow{a} (q'^S, q'^{TP}) \in T^{S\times TP} \leftrightarrow q^S \xrightarrow{a} q'^S \in T^S \bigwedge q^{TP} \xrightarrow{a} q'^{TP} \in T^{TP}$.
- $Accept^{SP}$ and $Refuse^{TP}$ are defined as follows:
  $Accept^{SP} = Q^{SP} \bigcap (Q^S \times Accept^{TP})$,
  $Refuse^{SP} = Q^{SP} \bigcap (Q^S \times Refuse^{TP})$,

The synchronous product marks behaviors of S by *Accept* and *Refuse*, and possibly unfold S, i.e. accepted behaviors of SP are exactly those behaviors of S which are accepted by TP.

3. The visible behavior is built from SP and then a Complete Test Graph (CTG) is built by extracting the accepted behaviors. Since IOLTSs differentiate between the input and output actions, TGV use this differentiation as controllability options to produce controllable or uncontrollable (complete) test cases. To be successful, the test cases must be controllable [24]. A test case is said to be controllable if at any state no choice need to be made between several outputs or output and inputs. These controllability options are:

   - Produce the complete test case: When this option is chosen, TGV produces the complete test graph without selecting a single test case from it. Possibly, this test case is not controllable. RGV defines some priority settings upon which a test case can be generated. The default priority of TGV is laid on input actions and on actions of the specification with

the possibility to give priority to test purpose actions or give priority to output actions. In fact, we wanted to know if there is an effect on the size of generated test case. We analyzed the effect of changing the priority settings in an experiment based on the RCCS case study illustrated in Figure 4.60 using the same combination of specification, test purpose, and input and output action and just varying the priority setting. Our experiment has the results for the number of states and transitions in the generated test cases shown in Table 5.2. From this table, we can induce that setting the priority only effects the selection of single test cases and has no effect on the generation of the complete test graph.

- Produce a controllable test case with loops: When this option is chosen, a single test case possibly containing loops will be generated. We used the test purpose in Table 5.1 to produce a test case from the example in Figure 4.60 using this option we obtain the test case shown in Figure 5.5.
- Produce a controllable test case without loops: When this option is chosen, TGV produces a single test case without loops. We used the test purpose in Table 5.1 also to produce a test case from the example in Figure 4.60 using this option. We happened to obtain the same test case as in the previous option test case shown in Figure 5.5.

des (0, 13, 10)
(0, "E1 !TRUE !TRUE", 2)
(2, "E1 !TRUE !TRUE", 3)
(3, "E1 !TRUE !TRUE", 1)
(0, "E1 !TRUE !TRUE", 4)
(4, "E1 !TRUE !TRUE", 1)
(2, "E2 !TRUE !TRUE", 5)
(5, "E2 !TRUE !TRUE", 6)
(6, "E2 !TRUE !TRUE", 1)
(2, "E3 !TRUE !TRUE", 7)
(7, "E2 !TRUE !TRUE", 8)
(8, "E3 !TRUE !TRUE", 9)
(9, "E2 !TRUE !TRUE", 1)
(1, "ACCEPT", 1)

Table 5.1: Test Purpose for the Example in Figure 4.60

Table 5.2: Complete Uncontrollable Test Cases

Priority Setting on	Number of states	Number of Transitions
specification/input	2851	6168
test purpose/input	2851	6168
specification/output	2851	6168
test purpose/output	2851	6168

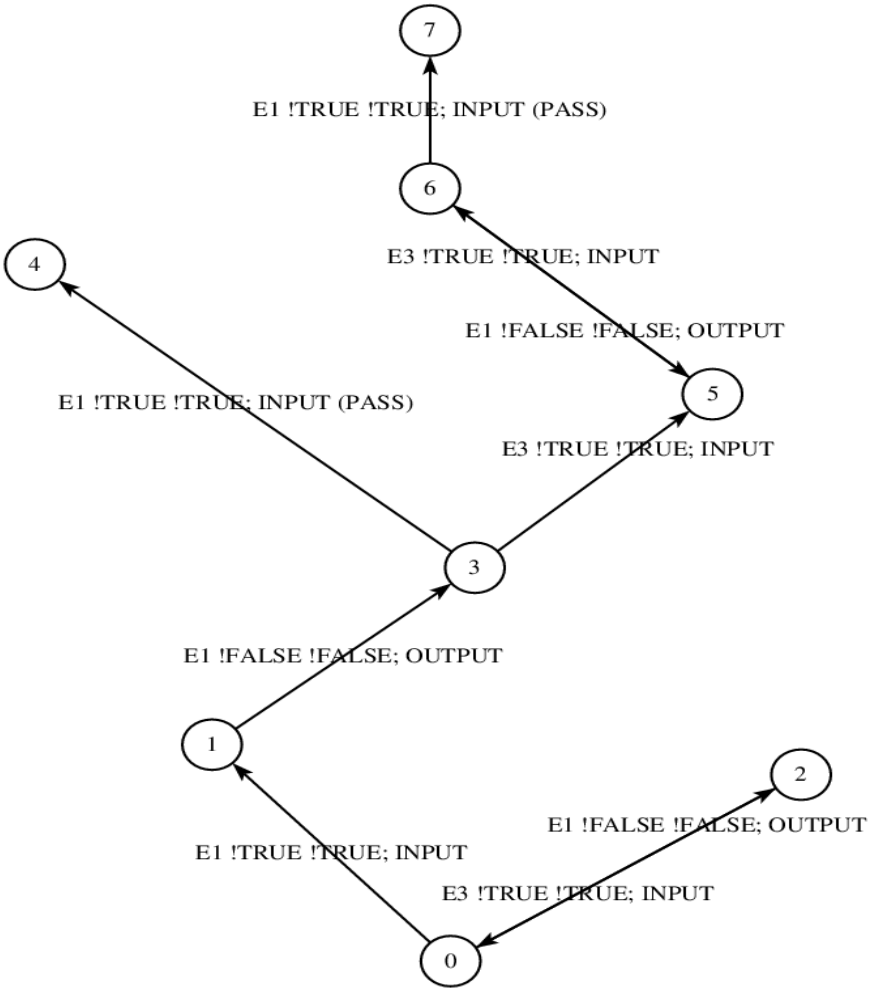

Figure 5.5: Sample Test Case (CADP Produced Graph)

Chapter 6

Conclusion

This dissertation proposed an approach for testing of safety-critical systems. It is based on integrating behavioral (CEFSMs are used to model a system behavior) and fault models (Fault tree is used to describe a failure). While one might be tempted to skip FTA and include the fault information ad hoc in the CEFSM directly, this is unsystematic and error prone. It also fails to provide a proper FTA, an important part of developing safety-critical systems. Since the behavioral models are normally designed by software engineer teams and the fault models are designed by safety engineer teams, some compatibility issues will most likely arise because of each team's perspective of the system. Examples of these compatibility issues are the naming differences of the events, or conditions for event occurrence. The two models are analyzed for compatibility and necessary changes are identified to make them compatible.

Due to these compatibility issues, it is necessary to perform a compatibility transformation to link the events at the behavioral model that contribute to a failure to those events at the fault tree that have the same meaning although they may have different naming methodologies. This is done by creating a class diagram for each entity or event that contributes to the failure. The output of this step is an FT',

the original FT expressed in terms of the compatible events and their conditions if any, i.e. *BFClass.BFEventCondition*.

After the compatibility transformation step is completed, we transform the gates of the fault tree into what we call gate CEFSM (GCEFSM) according to a set of transformation rules. We defined a GCEFSM for each fault tree gate such as: AND gate and OR gate,...etc. The fault tree is traversed and transformed gate by gate. At the end of this step, we obtain a transformed FT in a form of CEFSMs called GCEFSMs. The complete GCEFSMs is a collection of connected GCEFSMs that is equivalent to the original fault tree. We do this transformation to be able to integrate it with the behavioral model which is also in a CEFSM model. The integration is done by mapping the events from the behavioral model to those have the same meaning at the fault model. The resultant model is an integrated Communicating extended finite state machines ICEFSMs that is composed of the the MB and FM. Several coverage criteria besides those for the conventional graph where it is treated sequentially [4], are proposed for the integrated model. These criteria are meant to deal with the so-called rendezvous graph [151] of communicating processes. The rendezvous graph is the graph that contains the nodes involved in the communication between the communicating processes. These criteria focus on the global view of the integrated model.

Integrating mitigation models into ICEFSM in order to be used for testing proper mitigation is impractical. Clearly, it is hard to determine in which state the system is when the failure occurs. In other words, an event at state s_i may contribute in a failure, but the failure shows up when the system is in another state S_j. For that reason, we found that integrating mitigation models in every suspected state would produce a very large and complex system and hence we proposed using our approach as the first phase of an End-to-End safety-critical system testing methodology [6]

in which we can test proper safety mitigations. The first phase which is part of this dissertation, produces test cases from the integrated model that are then used to construct the applicability matrix. The applicability matrix is a two dimensional array where each row is a different failure type and each column is a behavioral state of the system. When a failure f_i occurs or is applicable in a state s_j, we put 1 in the position (f_i, s_j) and the rest of the matrix is filled with zeros. The second phase, which is part of the dissertation of Mrs. Salwa Elakeili, uses the applicability matrix to generate test cases for proper mitigation of failure according to several coverage criteria.

Model scalability is also investigated. To this end we developed a tool that estimates the number of states and transitions both for our approach and Sánchez *et al.*'s approach [127]. This tool integrates different behavioral model sizes with different fault trees sizes. We fed the tool with a variety of behavioral and fault models from relatively small models to big ones and let the tool compute the size of the integrated models. We then compared model sizes and investigated scalability. The variation of the model size gave us a clear idea of the growth of the integrated models for both approaches. We clearly showed that our approach is more scalable for all model sizes.

In this dissertation, we conducted three case studies with different sizes and from different application domains. The rationale behind choosing different model sizes from different domain is to show the applicability of our approach. We used a gas burner system (GBS), a relatively small example. The railroad crossing control system (RCCS) case study, a reasonably sized system, is also used. The third case study is a launch vehicle system (LVS) in which we integrated multiple fault trees. In these case studies we illustrated our approach step by step including the

compatibility transformation, model transformation, model integration, and test case generation.

CADP, a collection of analysis and testing tools, is used to analyze the integrated model. The integrated model is transformed into LOTOS format to be used as an input for CADP. Another tool was implemented to transform the integrated model into LOTOS. The tool transforms the behavioral model into LOTOS, transforms the fault tree into LOTOS and then integrates the two LOTOS models. The integrated LOTOS model is then given to CADP. CADP transforms the LOTOS into labeled transition systems (LTSs). Test generation with verification technology (TGV), a tool integrated to CADP, is used to generate test cases based on test purpose, coverage criteria, or mixture of both from the LTSs. The test cases can be presented in a form of a complete test graph (CTG). However, due to the huge number of the produced test cases in a reasonable system, displaying the CTG may not be feasible. CADP uses reachability analysis to find deadlock and live lock states in the model. Deadlock and livelock properties can be used to analyze the model.

The advantages of the proposed approach over those that deal with model integration are:

- This approach is capable of integrating more than one fault model with a behavioral model that contains a collection of processes.
- This approach is systematic as opposed to ad hoc.
- This approach is automated since it is algorithmic. We can build a tool that takes the behavioral model and fault models and does the integration. Moreover, the integrated model can be given to a tool to produce test cases.
- The integrated model is very concise compared to the EFSM approach [127] (*cf* section 4.1.1).

- Unlike the EFSM approach [127], this approach has the ability to model the whole model at once and does not need the minimum cut set.
- It uses explicit communicating edges, which makes it suitable for testing. Other approaches [41, 40, 82, 111] use implicit communication edges which makes them suitable for analysis not testing.

In summary we successfully provided a novel approach to test failures in safety critical systems that

- allows for systematic modeling and analysis for both functional and safety critical aspects of a system,
- fits into existing life cycles for developing SCSs,
- handles multiple Fault Trees,
- can be used as part of an end-to-end testing methodology,
- can be used with all existing test generation approach for CEFSMs, including [16, 86, 33, 68]
- can be used for model checking including deadlock and livelock,
- compares favorably to existing approaches whose scalability is limited, and
- applies to multiple application domains.

Chapter 7

Future Work

This work can be extended in a variety of ways:

- **Generalizability to other types of behavioral models:**

  We will investigate applying the integration approach for some other behavioral models that have the ability to describe communicating processes. Examples of such models are UML sequence and activity diagrams, Petri Nets. We think that integrating such models with failure models is straightforward since we have already defined the compatibility transformation between the behavioral and fault models in this approach and since these targeted models have the capability of modeling communicating processes. Generalizing this approach to other modeling languages will be of a great benefits since these modeling languages are used in other application domains, e. g. medical systems, robotic devices, and flight control systems.

- **Generalizability to other types of fault models:**

  In this approach we used fault trees as our fault model. However, using some other models that are used to model failure such as Event Tree Analysis (ET), Fault Hazard Analysis (FHA), and Failure Mode, Effects and Criticality

Analysis (FMECA) will be investigated. Some of these models can be used to determine the faults while other can be used for test case prioritization.

- **Application of the methodology to complex intelligent agents like unmanned vehicles and robots:**

  In this dissertation, we have shown that this approach works well for integrating communicating processes. We would like to apply this methodology in more complex intelligent agents where multiple independent agents are communicating to perform a certain task. The applications of intelligent agents have been used in safety-critical systems such as unmanned vehicle, unmanned planes, and robots in rescue missions in which the failure of the cooperation between a collection of robots may be catastrophic. Therefore, testing safety-critical behavior for such systems is essential. A failure that results from agent or communication malfunction can be described by a fault tree. Applying this approach for unmanned vehicle and multi-agent systems will give us more confidence that our approach is not only capable of handling complex systems, but it also shows further generalizability of the approach. For the unmanned vehicle, this research group is working towards modeling the environment and testing this system using CEFSMs[1]. Therefore, integrating fault models into such systems in order to test safety will be an important step to make these systems safer.

- **Experimental evaluation of the effectiveness of this approach:** At this point, our approach can be used to generate tests with a variety of existing test generation techniques such as [68, 16, 86] since the integrated models that our approach produces are CEFSMs that these techniques are capable of

[1]The Ph.D. dissertation of Mr. Mahmoud Abdelgawad.

testing. Hence, it is as effective as these techniques are. The effectiveness of this approach can be evaluated by producing concrete test cases from a model and execute them. Experiments can be designed to be used the proposed coverage criteria especially those that target the fault part of the model to see how effective our approach is.

Input space partitioning criteria [4] can also be used in conjunction with the proposed criteria for the existing testing techniques. The idea of fault injection can also be used to inject events as well as manipulating sensor values in order to target safety breaches. The test suite size, the percentage of failures captured, the time of the execution of the test suite are some of possible evaluation metrics.

Bibliography

[1] C. Abbaneo, F. Flammini, A. Lazzaro, P. Marmo, N. Mazzocca, and A. Sanseviero. UML Based Reverse Engineering for the Verification of Railway Control Logics. In *Dependability of Computer Systems, 2006. DepCos-RELCOMEX '06. International Conference on*, pages 3 –10, may 2006.

[2] A. Abdurazik and J. Offutt. Using UML Collaboration Diagrams for Static Checking and Test Generation. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 The Unified Modeling Language*, volume 1939 of *Lecture Notes in Computer Science*, pages 383–395. Springer Berlin / Heidelberg, 2000.

[3] P. Ammann, W. Ding, and D. Xu. Using a Model Checker to Test Safety Properties. In *Proceedings of the 7th International Conference on Engineering of Complex Computer Systems*, pages 212–221, Skövde, Sweden, 2001. IEEE.

[4] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[5] Paul Ammann and Jeff Offutt. *Introduction To Software Testing*. Cambridge University Press, 32 Avenue of the Americas, New York, NY 10013, USA, first edition, 2008.

[6] A. Andrews, A. Gario, and S. Elakeili. A Testing Methodology for Safety Critical Systems. *Software Testing, Verification and Reliability*, 2014, (Submitted).

[7] A.A. Andrews, J. Offutt, and R.T. Alexander. Testing Web Applications by Modeling with FSMs. *Software and Systems Modeling*, 4:326–345, 2005.

[8] Anneliese Andrews, Salwa Elakeili, and Salah Boukhris. Fail-safe test generation in safety critical systems. In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, pages 49–56. IEEE,

2014.

[9] J.D. Andrews and S.J. Dunnett. Event-Tree Analysis Using Binary Decision Diagrams. *IEEE Transactions on Reliability*, 49(2):230–238, Jun. 2000.

[10] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing,*, 1(1):11–33, 2004.

[11] J.C. Becker and G. Flick. A Practical Approach to Failure Mode, Effects and Criticality Analysis (FMECA) for Computing Systems. In *High-Assurance Systems Engineering Workshop, 1996. Proceedings., IEEE*, pages 228 –236, Oct. 1996.

[12] B. Berenbach and T. Wolf. A Unified Requirements Model: Integrating Features, Use Cases, Requirements, Requirements Analysis and Hazard Analysis. In *Second IEEE International Conference on Global Software Engineering, 2007. ICGSE 2007*, pages 197–203, Aug. 2007.

[13] A. Bertolino, E. Marchetti, and H. Muccini. Introducing a Reasonably Complete and Coherent Approach for Model-based Testing. *Electronic Notes in Theoretical Computer Science*, 116(0):85–97, 2005.

[14] A. Bertolino, E. Marchetti, and A. Polini. Integration of Components to Test Software Components. *Electronic Notes in Theoretical Computer Science*, 82(6):44–54, 2003.

[15] S. Boroday, A. Petrenko, R. Groz, and Y.M. Quemener. Test Generation for CEFSM Combining Specification and Fault Coverage. In *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*, TestCom '02, pages 355–372, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[16] C. Bourhfir, E. Aboulhamid, R. Dssouli, and N. Rico. A Test Case Generation

Approach for Conformance Testing of SDL Systems. *Comp. Commun.*, 24(3-4):319–333, 2001.

[17] C. Bourhfir, R. Dssouli, E.M. Aboulhamid, and N. Rico. A Guided Incremental Test Case Generation Procedure for Conformance Testing for CEFSM Specified Protocols. In *Proceedings of the IFIP TC6 11th International Workshop on Testing Communicating Systems*, IWTCS, pages 275–290, Deventer, The Netherlands, The Netherlands, 1998. Kluwer, B.V.

[18] J. Bowen and V. Stavridou. Safety-Critical Systems, Formal Methods and Standards. *Software Engineering Journal*, 8(4):189–209, July 1993.

[19] J.B. Bowles. An Assessment of RPN Prioritization in a Failure Modes Effects and Criticality Analysis. In *Annual Reliability and Maintainability Symposium, 2003*, pages 380–386, 2003.

[20] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, April 1983.

[21] L. C. Briand, J. Cui, and Y. Labiche. Towards Automated Support for Deriving Test Data from UML Statecharts. In *UML*, pages 249–264, 2003.

[22] L.C. Briand and Y. Labiche. A UML-Based Approach to System Testing. In *UML*, pages 194–208, 2001.

[23] R.C. Bromley and E. Bottomley. Failure Modes, Effects and Criticality Analysis (FMECA). In *IEEE Colloquium on Masterclass in Systems Engineering - Part Two*, pages 1–7, 1994.

[24] J.R. Calamé. Specification-based test generation with tgv, 2005.

[25] E.G. Cartaxo, F.G.O. Neto, and P.D.L. Machado. Test case generation by means of UML sequence diagrams and labeled transition systems. In *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, pages 1292–1297, 2007.

[26] A. Causevic, D. Sundmark, and S. Punnekkat. An Industrial Survey on Contemporary Aspects of Software Testing. In *Third International Conference on Software Testing, Verification and Validation (ICST 2010)*, pages 393–401, April 2010.

[27] M. Chen, X. Qiu, and X. Li. Automatic Test Case Generation for UML Activity Diagrams. In *AST*, pages 2–8, 2006.

[28] S.K. Chen, T.K. Ho, and B.H. Mao. Reliability Evaluations of Railway Power Supplies by Fault-Tree Analysis. *IET Electric Power Applications*, 1(2):161–172, March 2007.

[29] T.S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[30] M.F. Chudleigh. Hazard Analysis of a Computer Based Medical Diagnostic System. *Computer Methods and Programs in Biomedicine*, 44(1):45–54, 1994.

[31] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, and B.M. Horowitz. Model-Based Testing in Practice. In *ICSE*, pages 285–294, 1999.

[32] M.A. de Miguel, J.F. Briones, J.P. Silva, and A. Alonso. Integration of Safety Analysis in Model-Driven Software Development. *IET Software*, 2(3):260–280, June 2008.

[33] K. Dederian, R.M. Hierons, M. Harman, and Q. Guo. Input Sequence Generation for Testing of Communicating Finite State Machines (CFSMs). In *Proceedings of the 2004 Conference on Genetic and Evolutionary Computation (GECCO '04)*, pages 1429–1430, Seattle, Washington, USA, 26-30 June 2004.

[34] K. Derderian, R. Hierons, M. Harman, and Q. Guo. Estimating the Feasibility of Transition Paths in Extended Finite State Machines. *Automated Software*

Engineering, 17:33–56, 2010.

[35] G. Despotou, R. Alexander, and T. Kelly. Addressing Challenges of Hazard Analysis in Systems of Systems. In *2009 3rd Annual IEEE Systems Conference*, pages 167–172, March 2009.

[36] T. Dinh-Trong, N. Kawane, S. Ghosh, R. France, and A.A. Andrews. A tool-supported approach to testing UML design models. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, 2005. ICECCS 2005*, pages 519–528, 2005.

[37] R. Donini, S. Marrone, N. Mazzocca, A. Orazzo, D. Papa, and S. Venticinque. Testing Complex Safety-Critical Systems in SOA Context. In *International Conference on Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008.*, pages 87–93, Los Alamitos, CA, USA, Mar. 2008.

[38] J.B. Dugan, K.J. Sullivan, and D. Coppit. Developing a Low-Cost High-Quality Software Tool for Dynamic Fault-Tree Analysis. *IEEE Transactions on Reliability*, 49(1):49–59, Mar. 2000.

[39] J. Dunjó, V. Fthenakis, J.A. Vlchez, and J. Arnaldos. Hazard and Operability (HAZOP) Analysis. A literature Review. *Journal of Hazardous Materials*, 173(13):19–32, 2010.

[40] O. El Ariss, D. Xu, and W.E. Wong. Integrating Safety Analysis With Functional Modeling. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 41(4):610 –624, July 2011.

[41] O. El Ariss, D. Xu, W.E. Wong, Y. Chen, and Y. Lee. A Systematic Approach for Integrating Fault Trees into System Statecharts. In *32nd Annual IEEE International on Computer Software and Applications, 2008. COMPSAC '08.*, pages 120–123, Turku, Finland, Aug. 2008.

[42] S.L.R. Ellison and V.J. Barwick. Estimating Measurement Uncertainty: Rec-

onciliation Using a Cause and Effect Approach. *Accreditation and Quality Assurance: Journal for Quality, Comparability and Reliability in Chemical Measurement*, 3:101–105, 1998.

[43] C.A. Ericson. *Hazard Analysis Techniques for System Safety.* Wiley-Interscience, 2005.

[44] M. Fantinato and M. Jino. Applying Extended Finite State Machines in Software Testing of Interactive Systems. In Joaquim Jorge, Nuno Jardim Nunes, and João Falcão e Cunha, editors, *Interactive Systems. Design, Specification, and Verification*, volume 2844 of *Lecture Notes in Computer Science*, pages 109–131. Springer Berlin / Heidelberg, 2003.

[45] Francesco Flammini, Nicola Mazzocca, and Antonio Orazzo. Automatic Instantiation of Abstract Tests on Specific Configurations for Large Critical Control Systems. *Software Testing, Verification & Reliability*, 19:91–110, June 2009.

[46] Martin Fowler. *UML Distilled.* Pearson Addison Wesley, 501 Boylston St., Suite 900, Boston, Massachusetts 02116, USA, third edition, 2004.

[47] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected State Machine Coverage for Software Testing. *SIGSOFT Software Engineering Notes*, 27(4):134–143, Jul. 2002.

[48] S. Fujiwara, G. von Bochmann, F.Khendek, M. Amalou, and A. Ghedamsi. Test Selection Based on Finite State Models. *IEEE Transactions on Software Engineering*, SE-17(6):591–603, June 1991.

[49] Hubert Garavel, Claude Helmstetter, Olivier Ponsini, and Wendelin Serwe. Verification of an industrial systemc/tlm model using lotos and cadp. In *MEMOCODE*, pages 46–55, 2009.

[50] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp

2011: a toolbox for the construction and analysis of distributed processes. *The International Journal on Software Tools for Technology Transfer (STTT)*, 15(2):89–107, 2013.

[51] Hubert Garavel, Radu Mateescu, and Wendelin Serwe. Large-scale Distributed Verification Using CADP: Beyond Clusters to Grids. *Electronic Notes Theory Computer Science*, 296:145–161, 2013.

[52] A. Gario and A. Andrews. Fail-Safe Testing of Safety-Critical Systems. In *Software Engineering Conference (ASWEC), 2014 23rd Australian*, pages 190–199. IEEE, 2014.

[53] A. Gario and A. Andrews. Fail-Safe Testing of Safety-Critical Systems:A Case Study and Efficiency Analysis. *Software Quality Journal*, 2014, (Submitted).

[54] A. Gario, A. Andrews, and S. Hagerman. Testing of safety-critical systems: An aerospace launch application. In *Aerospace Conference, 2014 IEEE*, pages 1–17. IEEE, 2014.

[55] S. Gnesi, D. Latella, and M. Massink. Formal Test-Case Generation for UML Statecharts. In *ICECCS*, pages 75–84, 2004.

[56] P.L. Goddard. Validating the Safety of Embedded Real-Time Control Systems Using FMEA. In *Reliability and Maintainability Symposium, 1993. Proceedings., Annual*, pages 227 –230, Jan. 1993.

[57] M. Gogolla, J. Bohling, and M. Richters. Validation of UML and OCL Models by Automatic Snapshot Generation. In *UML*, pages 265–279, 2003.

[58] L.D. Gowen, J.S. Collofello, and F.W. Calliss. Preliminary Hazard Analysis for Safety-Critical Software Systems. In *Eleventh Annual International Phoenix Conference on Computers and Communications, 1992. Conference Proceedings.*, pages 501–508, Apr. 1992.

[59] F. Gregory. Cause, Effect, Efficiency and Soft Systems Models. *The Journal*

of the Operational Research Society, 44(4):pp. 333–344, Apr. 1993.

[60] J. Groβmann, P. Makedonski, H.W. Wiesbrock, J. Svacina, I. Schieferdecker, and J. Grabowski. Model-Based X-in-the-Loop Testing. In *Model-Based Testing for Embedded Systems*, pages 299–335. CRC Press, 2011.

[61] G. Di Guglielmo, F. Fummi, C. Marconcini, and G. Pravadelli. A Pseudo-Deterministic Functional ATPG based on EFSM Traversing. In *Sixth International Workshop on Microprocessor Test and Verification, 2005. MTV '05.*, pages 70–75, Nov. 2005.

[62] G.D. Guglielmo, L.D. Guglielmo, F. Fummi, and G. Pravadelli. Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs. *Journal of Electronic Testing*, 27:137–162, April 2011.

[63] F. Hadipriono, C. Lim, and K. Wong. Event Tree Analysis to Prevent Failures in Temporary Structures. *Journal of Construction Engineering and Management*, 112(4):500–513, 1986.

[64] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-Based Integration Testing. *SIGSOFT Software Engineering Notes*, 25(5):60–70, Aug. 2000.

[65] M. Hause, A. Stuart, D. Richards, and J. Holt. Testing Safety Critical Systems with SysML/UML. In *2010 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS).*, pages 325–330, Mar. 2010.

[66] H. Hecht, X. An, and M. Hecht. Computer Aided Software FMEA for Unified Modeling Language Based Software. In *Reliability and Maintainability, 2004 Annual Symposium - RAMS*, pages 243–248, 2004.

[67] O. Henniger, M. Lu, and H. Ural. Automatic Generation of Test Purposes for Testing Distributed Systems. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing*, volume 2931 of *Lecture Notes*

in Computer Science, pages 1105–1105. Springer Berlin/Heidelberg, 2004.

[68] A. Hessel and P. Pettersson. A Global Algorithm for Model-Based Test Suite Generation. *Electronic Notes in Theoretical Computer Science*, 190(2):47–59, 2007.

[69] E.J. Hill and L.J. Bose. Sneak Circuit Analysis of Military Systems. In *Proceedings of the Second International System Safety Conference*, pages 351–372, July 1975.

[70] W.E. Howden. Methodology for the Generation of Program Test Data. *IEEE Transactions on Computers*, C-24(5):554–560, May 1975.

[71] Su-Yu Hsu and Chyan-Goei Chung. A heuristic approach to path selection problem in concurrent program testing. In *Proceedings of the Third Workshop on Future Trends of Distributed Computing Systems, 1992*, pages 86–92, Apr 1992.

[72] J.C. Huang. An Approach to Program Testing. *ACM Comput. Surv.*, 7(3):113–128, Sep. 1975.

[73] J. Hwang, H. Jo, and D. Kim. Hazard Analysis of Train Control System Using HAZOP-KR Methods. In *2010 International Conference on Electrical Machines and Systems (ICEMS 2010)*, pages 1971–1975, Oct. 2010.

[74] Claude Jard and Thierry Jéron. TGV: Theory, Principles and Algorithms: A Tool for the Automatic Synthesis of Conformance Test Cases for Non-deterministic Reactive Systems. *International Journal on Software Tools Technology Transfer*, 7(4):297–315, August 2005.

[75] B. Kaiser. A Fault-Tree Semantics to Model Software-Controlled Systems. *Softwaretechnik-Trends*, 23(3):33–39, 2003.

[76] B. Kaiser. Extending the Expressive Power of Fault Trees. In *Proceedings on Reliability and Maintainability Symposium, 2005*, pages 468–474, Alexandria,

Virginia, USA, 2005.

[77] B. Kaiser, P. Liggesmeyer, and O. Mäckel. A New Component Concept for Fault trees. In *Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software*, volume 33 of *SCS '03*, pages 37–46, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[78] A. Kalaji, R.M. Hierons, and S. Swift. A Search-Based Approach for Automatic Test Generation from Extended Finite State Machine (EFSM). In *Testing: Academic and Industrial Conference - Practice and Research Techniques, 2009. TAIC PART '09.*, pages 131–132, Sept. 2009.

[79] S. Kaplan. Matrix Theory Formalism for Event Tree Analysis: Application to Nuclear-Risk Analysis. *Risk Analysis*, 2(1):9–18, 1982.

[80] R.M. Keller. Formal Verification of Parallel Programs. *Communications of the ACM*, 19(7):371–384, 1976.

[81] R.S. Kenett. *Cause-and-Effect Diagrams*, chapter 3. John Wiley & Sons, Ltd, 2008.

[82] H. Kim, W.E. Wong, V. Debroy, and D. Bae. Bridging the Gap between Fault Trees and UML State Machine Diagrams for Safety Analysis. In *17th Asia Pacific Software Engineering Conference (APSEC)*, pages 196–205, 30-Dec. 2010.

[83] T.A. Kletz. *Hazop and Hazan*. IChemE, 2006.

[84] J. Kloos and R. Eschbach. A Systematic Approach to Construct Compositional Behaviour Models for Network-structured Safety-critical Systems. *Electronic Notes Theoretical Computer Science*, 263:145–160, June 2010.

[85] J. Kloos, T. Hussain, and R. Eschbach. Risk-Based Testing of Safety-Critical Embedded Systems Driven by Fault Tree Analysis. In *IEEE International Conference on Software Testing Verification and Validation Workshop*

(ICSTW 2011), volume 0, pages 26–33, Los Alamitos, CA, USA, Mar. 2011. IEEE Computer Society.

[86] G. Kovács, Z. Pap, and G. Csopaki. Automatic Test Selection Based on CEFSM Specifications. *Acta Cybern.*, 15(4):583–599, Dec. 2002.

[87] D. Latella and M. Massink. A formal testing framework for UML statechart diagrams behaviors: from theory to automatic verification. In *Sixth IEEE International Symposium on High Assurance Systems Engineering*, pages 11–22, 2001.

[88] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines- A Survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[89] R. Lefticaru and F. Ipate. Automatic State-Based Test Generation Using Genetic Algorithms. In *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2007. SYNASC.*, pages 188–195, Sept. 2007.

[90] Barbara Staudt Lerner, Stefan Christov, Leon J. Osterweil, Reda Bendraou, Udo Kannengiesser, and Alexander Wise. Exception Handling Patterns for Process Modeling. *IEEE Transactions on Software Engineering*, 36(2):162–183, 2010.

[91] N.G. Leveson and P.R. Harvey. Analyzing Software Safety. *IEEE Transactions on Software Engineering*, SE-9(5):569–579, Sept. 1983.

[92] Bao-Lin Li, Zhi-shu Li, Li Qing, and Yan-Hong Chen. Test Case Automate Generation from UML Sequence Diagram and OCL Expression. In *Proceedings of the 2007 International Conference on Computational Intelligence and Security*, CIS '07, pages 1048–1052, Washington, DC, USA, 2007. IEEE Computer Society.

[93] J.J. Li and W.E. Wong. Automatic Test Generation From Communicating

Extended Finite State Machine (CEFSM)-Based Models. In *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002. (ISORC 2002). Proceedings*, pages 181–185, 2002.

[94] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang. Generating Test Cases from UML Activity Diagram Based on Gray-Box Method. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 284–291, Washington, DC, USA, 2004. IEEE Computer Society.

[95] W.-C. Liu and C.-G. Chung. Symbolic Path-based Protocol Verification. *Information & Software Technology*, 42(4):245–255, 2000.

[96] M. Lochau and U. Goltz. Feature Interaction Aware Test Case Generation for Embedded Control Systems. *Electron. Theory Computer Science Notes*, 264:37–52, Dec. 2010.

[97] L. Lucio, L. Pedro, and D. Buchs. A Methodology and a Framework for Model-Based Testing. In Nicolas Guelfi, editor, *Rapid Integration of Software Engineering Techniques*, volume 3475 of *Lecture Notes in Computer Science*, pages 619–619. Springer Berlin/Heidelberg, 2005.

[98] G. Luo, R. Dssouli, G. von Bochmann, P. Venkataram, and A. Ghedamsi. Generating Synchronizable Test Sequences Based on Finite State Machine with Distributed Ports. In *Protocol Test Systems*, pages 139–153, 1993.

[99] R. Manian, J.B. Dugan, D. Coppit, and K.J. Sullivan. Combining Various Solution Techniques for Dynamic Fault Tree Analysis of Computer Systems. In *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium, 1998*, pages 21–28, Nov. 1998.

[100] D.A. Mathaikutty, S. Ahuja, A. Dingankar, and S. Shukla. Model-Driven Test Generation for System Level Validation. In *IEEE International on High Level*

Design Validation and Test Workshop, 2007. (HLVDT 2007), pages 83–90, Nov. 2007.

[101] Ben Swarup Medikonda and P. Seetha Ramaiah. Integrated safety analysis of software-controlled critical systems. *SIGSOFT Softw. Eng. Notes*, 35(1):1–7, January 2010.

[102] Sandfoss R. Meyer S. Applying Use-Case Methodology to SRE and System Testing. In *STAR West Conference*, SWC '98, 1998.

[103] P.V.R. Murthy, P.C. Anitha, M. Mahesh, and R. Subramanyan. Test Ready UML Statechart Models. In *Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, SCESM '06, pages 75–82, New York, NY, USA, 2006. ACM.

[104] Ashalatha Nayak and Debasis Samanta. Automatic Test Data Synthesis using UML Sequence Diagrams. *Journal of Object Technology*, pages 115–144, 2010.

[105] R. Nazier and T. Bauer. Automated Risk-Based Testing by Integrating Safety Analysis Information into System Behavior Models. In *IEEE 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW), 2012*, pages 213–218, 2012.

[106] G. De Nicola, P. di Tommaso, E. Rosaria, F. Francesco, M. Pietro, and O. Antonio. A Grey-Box Approach to the Functional Testing of Complex Automatic Train Protection Systems. In Mario Dal Cin, Mohamed Kaniche, and Andros Pataricza, editors, *Dependable Computing - EDCC 2005*, volume 3463 of *Lecture Notes in Computer Science*, pages 305–317. Springer Berlin / Heidelberg, 2005.

[107] B. Nielsen and A. Skou. Automated Test Generation from Timed Automata. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 343–357,

London, UK, 2001. Springer-Verlag.

[108] A.J. Offutt, Y. Xiong, and S. Liu. Criteria for Generating Specification-based Tests. In *Fifth IEEE International Conference on Engineering of Complex Computer Systems, 1999. ICECCS '99*, pages 119–129, 1999.

[109] J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. In Robert France and Bernhard Rumpe, editors, *UML99 The Unified Modeling Language*, volume 1723 of *Lecture Notes in Computer Science*, pages 76–76. Springer Berlin / Heidelberg, 1999.

[110] J. Offutt, S. Liu, A. Abdurazik, and P Ammann. Generating Test Data from State-Based Specifications. *Software Testing, Verification & Reliability*, 13(1):2553, 2003.

[111] F. Ortmeier, M. Güdemann, and R. Wolfgang. Formal Failure Models. In *Proceedings of the 1st IFAC Workshop on Dependable Control of Discrete Systems (DCDS 07)*. Elsevier, 2007.

[112] F. Ortmeier and G. Schellhorn. Formal Fault Tree Analysis - Practical Experiences. *Electronic Notes in Theoretical Computer Science*, 185:139–151, Jul. 2007.

[113] F. Ouabdesselam and I. Parissis. Testing Synchronous Critical Software. In *Proceedings of the 5th International Symposium on Software Reliability Engineering, 1994*, pages 239–248, Monterey, California, Nov. 1994.

[114] N. Ozarin. Failure Modes and Effects Analysis During Design of Computer Software. In *Reliability and Maintainability, 2004 Annual Symposium - RAMS*, pages 201–206, 2004.

[115] C.S. Pasareanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karsai, H. Nine, and S. Neema. Model Based Analysis and Test Generation for Flight Software. In *Proceedings of the Third IEEE International Conference on Space Mission*

Challenges for Information Technology, pages 83–90, Washington, DC, USA, 2009. IEEE Computer Society.

[116] T. Pasquale, E. Rosaria, M. Pietro, O. Antonio, and A.S. Ferroviario. Hazard Analysis of Complex Distributed Railway Systems. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems, 2003*, pages 283–292, Florence, Italy, Oct. 2003.

[117] T. Pender. *UML Bible*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003.

[118] A. Petrenko, S. Boroday, and R. Groz. Confirming Configurations in EFSM Testing. *IEEE Transactions on Software Engineering*, 30(1):29–42, Jan. 2004.

[119] Antonella Petrillo, Roberta Fusco, Vincenza Granata, Salvatore Filice, Nicola Raiano, Daniela Maria Amato, Maria Zirpoli, Alessandro di Finizio, Mario Sansone, Anna Russo, et al. Risk Management in Magnetic Resonance: Failure Mode, Effects, and Criticality Analysis. *BioMed research international*, 2013, 2013.

[120] O. Pilskalns, A. A. Andrews, S. Ghosh, and R. B. France. Rigorous Testing by Merging Structural and Behavioral UML Representations. In *UML*, pages 234–248, 2003.

[121] O. Pilskalns, G. Uyan, and A. Andrews. Regression Testing UML Designs. In *22nd IEEE International Conference on Software Maintenance, 2006. ICSM '06.*, pages 254–264, 2006.

[122] M. Prăsănna and K.R. Chandran. Automatic Test Case Generation for UML Object Diagrams Using Genetic Algorithm. *International Journal of Soft Computing Applications*, 1(1):19–32, July 2009.

[123] D.J. Reifer. Software Failure Modes and Effects Analysis. *IEEE Transactions on Reliability*, R-28(3):247–249, Aug. 1979.

[124] H. Reza, S. Buettner, and V. Krishna. A Method to Test Component Off-the-Shelf (COTS) Used in Safety Critical Systems. In *Fifth International Conference on Information Technology: New Generations, 2008. ITNG 2008.*, pages 189–194, Las Vegas, Nevada, USA, Apr. 2008.

[125] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Boston, MA, 2005.

[126] M. Sánchez, J.C. Augusto, and M. Felder. Fault-based Testing of E-Commerce Applications. In *Proceedings of the 2nd International Workshop on Verification and Validation of Enterprise Information Systems*, pages 66–72, 2004.

[127] M. Sánchez and M. Felder. A Systematic Approach to Generate Test Cases based on Faults. In *Argentine Symposium in Software Engineering*, Buenos Aires, Argentina, 2003.

[128] Monalisa Sarma, P. V. R. Murthy, Sylvia Jell, and Andreas Ulrich. Model-based testing in industry: a case study with two MBT tools. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 87–90, New York, NY, USA, 2010. ACM.

[129] Devon Simmonds, Arnor Solberg, Raghu Reddy, Robert France, and Sudipto Ghosh. An Aspect Oriented Model Driven Framework. In *EDOC Enterprise Computing Conference, 2005 Ninth IEEE International*, pages 119–130. IEEE, 2005.

[130] A.E. Summers. Introduction to Layers of Protection Analysis. *Journal of Hazardous Materials*, 104(13):163–168, 2003.

[131] H. Sun, M. Hauptman, and R. Lutz. Integrating Product-Line Fault Tree Analysis into AADL Models. In *High Assurance Systems Engineering Symposium, 2007. HASE '07. 10th IEEE*, pages 15–22, Nov. 2007.

[132] S. Supakkul and L. Chung. Applying a Goal-Oriented Method for Hazard

Analysis: A Case Study. In *Fourth International Conference on Software Engineering Research, Management and Applications, 2006.*, pages 22–30, Aug. 2006.

[133] S.K. Swain and D.P. Mohapatra. Test Case Generation from Behavioral UML Models. *International Journal of Computer Applications*, 6(8):080000–11, Sep. 2010.

[134] L.H. Tahat, B. Vaysburg, B. Korel, and A.J. Bader. Requirement-Based Automated Black-Box Test Generation. In *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*, pages 489–495, 2001.

[135] Richard N. Taylor. A General-Purpose Algorithm for Analyzing Concurrent Programs. *Commun. ACM*, 26(5):361–376, May 1983.

[136] N. Tracey, J. Clark, J. Mcdermid, and K. Mander. Integrating Safety Analysis with Automatic Test-Data Generation for Software Safety Verification. In *Proceedings of 17th International System Safety Conference (ISSC 1999)*, pages 128–137, Orlando, FL, USA, 1999. System Safety Society.

[137] Jan Tretmans. Model Based Testing with Labeled Transition Systems. In *Formal Methods and Testing*, pages 1–38, 2008.

[138] B-Y. Tsai, S. Stobart, N. Parrington, and I. Mitchell. An Automatic Test Case Generator Derived from State-based Testing. In *Software Engineering Conference, 1998. Proceedings*, pages 270–277, 1998.

[139] M. Utting. How to Design Extended Finite State Machine Test Models in Java. In *Model-Based Testing for Embedded Systems*, pages 147–169. CRC Press, 2011.

[140] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-based Testing Approaches. *Software Testing, Verification and Reliability*, 22(5):297–

312, 2012.

[141] J. Vain, A. Kull, M. Kaaramees, M. Markvardt, and K. Raiend. Reactive Testing of Nondeterministic Systems by Test Purpose-Directed Tester. In *Model-Based Testing for Embedded Systems*, pages 425–452. CRC Press, 2011.

[142] VASY. CADP (Caesar/Aldebaran Development Package). http://cadp.inria.fr/.

[143] W. Vesely, J. Dugan, J. Fragola, Minarick, and J. Railsback. Fault Tree Handbook with Aerospace Applications. Handbook, National Aeronautics and Space Administration, Washington, DC, 1981.

[144] J.M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[145] L. Wang and K.C. Tan. Software Testing for Safety Critical Applications. *Instrumentation Measurement Magazine, IEEE*, 8(2):38–47, June 2005.

[146] B.C. Wei. A Unified Approach to Failure Mode, Effects and Criticality Analysis (FMECA). In *Reliability and Maintainability Symposium, 1991. Proceedings., Annual*, pages 260–271, Jan. 1991.

[147] S. Weiβleder and H. Schlingloff. Automatic Model-Based Test Generation from UML State Machines. In *Model-Based Testing for Embedded Systems*, pages 77–109. CRC Press, 2011.

[148] S.N. Weiss. A Formal Framework for the Study of Concurrent Program Testing. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, 1988*, pages 106–113, Jul 1988.

[149] P.J. Wilkinson and T.P. Kelly. Functional Hazard Analysis for Highly Integrated Aerospace Systems. In *Certification of Ground/Air Systems Seminar (Ref. No. 1998/255), IEE*, pages 4/1 –4/6, Feb. 1998.

[150] Y. Wu, M.H. Chen, and J. Offutt. UML-Based Integration Testing for

Component-Based Software. In Hakan Erdogmus and Tao Weng, editors, *COTS-Based Software Systems*, volume 2580 of *Lecture Notes in Computer Science*, pages 251–260. Springer Berlin/Heidelberg, 2003.

[151] R.-D. Yang and Chyan-Goei Chung. A Path Analysis Approach to Concurrent Program Testing. In *Ninth Annual International Phoenix Conference on Computers and Communications, 1990. Conference Proceedings*, pages 425–432, Mar 1990.

[152] M. Zheng, V. Alagar, and O. Ormandjieva. Automated Generation of Test Sites from Formal Specifications of Real-Time Reactive Systems. *The Journal of System and Software*, 81:286–304, Feb. 2008.

[153] G. Zoughbi, L. Briand, and Y. Labiche. A UML Profile for Developing Airworthiness-Compliant (RTCA DO-178B), Safety-Critical Software. In Gregor Engels, Bill Opdyke, Douglas Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 574–588. Springer Berlin / Heidelberg, 2007.