View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Institute of Transport Research:Publications

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbeit
FACHBEREICH ELEKTROTECHNIK
UND INFORMATIONSTECHNIK

MASTER THESIS

Title: **Semantics of Non-Deterministic Repairable Fault Trees**

Submitted by: **Yogeswari Renganathan (Matr. No. 752471)**

1st Academic Supervisor: **Prof. Dr.-Ing Karl Kleinmann**

2nd Academic Supervisor: **Prof. Dr.-Ing Markus Haid**

Industrial Supervisor: **Sascha Müller**

Completion Date: **20.09.2019**

Student:

Yogeswari Renganathan
First (Given) Name Last (Family) Name

Date of Birth: 03.07.1991 Matr.-No.: 753471

1st Academic Supervisor: Prof. Dr.-Ing Karl Kleinmann

2nd Academic Supervisor: Prof. Dr.-Ing Markus Haid

Title: Semantics of Non-Deterministic Repairable Fault Trees

Abstract: (max 10 Lines)

Fault Tree Analysis is a popular technique used to support the design of critical systems. In a prior work, fault tree semantics have been developed for Non-Deterministic Dynamic Fault Trees that introduces non-determinism to the recovery actions to solve the problem of spare races and improve system reliability. However the existing work only deals with permanent faults. The focus of the thesis work is extending the formalism of Non-Deterministic Dynamic Fault Trees to support the notion of repair and develop semantics for Non-Deterministic Repairable Fault Trees to achieve higher availability of system. It includes formalizing the gate semantics and adapting the algorithms for analyzing the fault tree. Furthermore, the thesis work also adapts the minimization algorithms to produce a more compact version of the Recovery Automaton with fewer states.

In partial fulfilment of the requirements of the **University of Applied Sciences Hochschule Darmstadt (h_da)** for the degree **Master of Science in Electrical Engineering** carried out in collaboration with **Industrial Enterprise**

Company: German Aerospace Center (DLR), Simulations- und Softwaretechnik

Address: Lilienthalplatz 7, 38108 Braunschweig

This Master Thesis is subject to a non-disclosure agreement between the University of Applied Sciences Hochschule Darmstadt (h_da) and the industrial partner.

(Signature)

1st Academic Supervisor:

Student:

Yogeswari Renganathan
First (Given) Name Last (Family) Name

1st Academic Supervisor: Prof. Dr.-Ing Karl Kleinmann

2nd Academic Supervisor: Prof. Dr.-Ing Markus Haid

Declaration

I hereby declare that this thesis is a presentation of my original research work and that no other sources were used other than what is cited.
I furthermore declare that wherever contributions of others are involved, this contribution is indicated, clearly acknowledged and due reference is given to the author and source.
I also certify that all content without reference or citation contained in this thesis is original work.
I acknowledge that any misappropriation of the previous declarations can be considered a case of academic fraud.

Darmstadt, ...
(Date) (Signature)

Contents

List of Abbreviations

FDIR	Fault Detection, Isolation and Recovery
FTA	Fault Tree Analysis
FT	Fault Tree
SFT	Static Fault Tree
DFT	Dynamic Fault Tree
RFT	Repairable Fault Tree
RDFT	Repairable Dynamic Fault Tree
NDDFT	Non-Deterministic Dynamic Fault Tree
NDRFT	Non-Deterministic Repairable Fault Tree
PAND	Priority AND Gate
POR	Priority OR Gate
FDEP	Functional Dependency Gate
SPARE	Spare Gate
BE	Basic Event
TLE	Top Level Event
ES	Event Set
RES	Repairable Event Set
MA	Markov Automaton
RA	Recovery Automaton

List of Figures

List of Tables

Abstract

Fault Tree Analysis is a popular technique used to support the design of critical systems. In a prior work, fault tree semantics have been developed for Non-Deterministic Dynamic Fault Trees that introduces non-determinism to the recovery actions to solve the problem of spare races and improve system reliability. However the existing work only deals with permanent faults. The focus of the thesis work is extending the formalism of Non-Deterministic Dynamic Fault Trees to support the notion of repair and develop semantics for Non-Deterministic Repairable Fault Trees to achieve higher availability of system. It includes formalizing the gate semantics and adapting the algorithms for analyzing the fault tree. Furthermore, the thesis work also adapts the minimization algorithms to produce a more compact version of the Recovery Automaton with fewer states.

Chapter 1

Introduction

This chapter introduces the topic for the thesis work and explains the motivation and the expected goals of the thesis. An outline of the report is also provided at the end of the chapter.

Space missions and safety critical applications demand high reliability and fault tolerance. Spacecrafts and satellites can only be controlled in a limited way from ground stations due to the distance and time delay for sending signals. The spacecraft is exposed to the harsh conditions in space, resulting from radiations. The large distance between the spacecraft and ground maintenance also presents a challenge for the ground maintenance team to communicate with the system in a timely manner. Due to these challenges, on-board Fault Detection, Isolation and Recovery (FDIR) modules are needed to ensure stable operation of the space craft with minimal human interactions [WF13]. Even well-designed systems are not free of faults, but improving the fault tolerance of the system makes it more reliable. A fault tolerant system is a system that can preserve its operation without external assistance [Avi76]. For instance, a spacecraft uses redundancy by having duplicate critical components to ensure fault tolerance. The on-board Fault Detection, Isolation and Recovery (FDIR) modules are developed during the engineering phase of the spacecrafts and define how the system reacts to failures and the recovery actions to be taken [SD13].

Fault Tree Analysis is a technique that is used in industries to support designing FDIR concepts, which will be the focus of the thesis. Fault Tree Analysis (FTA) [Eri99] is an analysis technique that can model the possible faults of a system and evaluate the system to derive useful information like system vulnerability, system reliability and the most optimal recovery action that should be taken by the system. It helps to minimize the occurrence of faults and prepares the system in advance to different malfunctions that can occur in the given system. This thesis work develops a new semantic for fault tree which is the core element in FTAs for the application of modeling faults during the design phase of satellites.

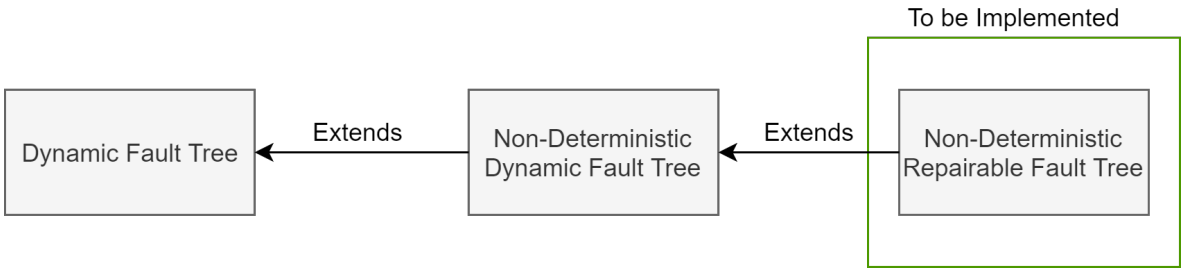


Figure 1.1: Abstract Representation of the Existing and To be implemented Semantics

1.1 Motivation

Fault Trees are the core elements used in FTA for modeling the faults of the given system. Different variants of Fault Trees that have different modeling capabilities have been researched upon in literature. In a prior research work at the German Aerospace Center (DLR), semantics for Non-Deterministic Dynamic Fault Trees [MGN18] have been developed. It is based on the Dynamic Fault Tree semantics and extends the formalism by introducing non-determinism for recovery actions concerning the spare resources. The optimal transitions are then chosen based on a scheduler to improve the reliability of the system. However, the formalism only considers permanent failures and does not capture transient or repairable faults that can occur in the system. Transient and repairable faults affect the actual availability of the system and thus need to be taken into consideration for system analysis. Though research on Repairable Fault Tree semantics and Repairable Dynamic Fault Trees (RDFT) have been done before, there is no existing work that combines the repair semantics with non-deterministic recovery actions and DFTs together. Combining the non-deterministic recovery action with repair, the Fault Tree Analysis technique is expected to achieve a higher availability for the system in comparison to RDFT. The focus of the thesis is to extend the Non-Deterministic Dynamic Fault Tree formalism and add support for modeling non-permanent faults by developing semantics for a repairable system and implement the concept in the Virtual Satellite framework. The Fig. 1.1 shows the representation of the existing work and the missing semantics that is to be implemented in this thesis.

1.2 Goals

The expected goals of this work have been defined below:

- Investigation of the state-of-the-art of Fault Trees and review of the variants of Fault Trees that have already been developed
- Development of a semantics for a Fault Tree model that combines the following flavors:

  - Non-deterministic recovery actions for spare resources
  - Dynamic behaviour of faults
  - Repairability of components in the system

- Adaption of the existing algorithms to generate Markov models from the Fault Trees based on the semantics developed for system analysis
- Extension of the Recovery Automaton model and synthesis method to include the notion of repair
- Adaption of the minimization algorithms to handle the state space explosion in Recovery Automata and produce a more compact version of the Recovery Automata.

1.3 Thesis Outline

The thesis starts with an introduction and follows up with Chapter. 2 providing an overview of the fundamentals and technical background relevant to the thesis work. An introduction to the preliminaries and a review of state-of-the-art in Fault Trees and relevant research work in the area of Fault Tree Analysis is discussed. The Chapter. 3 discusses the conceptual aspect of the proposed idea for the thesis. The implementation for the proposed concept is then discussed in Chapter. 4 which provides an overview of the architecture and algorithms adapted. The thesis then covers the evaluation and results for the work done in Chapter. 5 and finally based on the evaluation we conclude in the final Chapter. 6 by providing a summary and future outlook.

Chapter 2

Technical Background

This chapter provides an overview of the technical background and state of the art of the relevant work done in Fault Tree Analysis. It also describes the existing work on which the thesis was based on. The Chapter ends with a brief description of the list of existing tools for Fault Tree Analysis.

2.1 Fault Detection, Isolation and Recovery

Fault Detection, Isolation and Recovery (FDIR) is a mechanism used in safety-critical applications to maintain stable operation of the system even in the presence of faults. The FDIR module runs in parallel with the main system and tries to stabilize the system in case of malfunction or fault, with minimal external control. [SD13] While faults are deviations of at least one property that may lead to malfunctions, partial or total failure, failures are defined as total shutdown of the system or subsystem under consideration [WF13]. Not all faults lead to failure and this require appropriate FDIR techniques for ensuring stable operation of the system. Especially in space applications where there might be delay in fault diagnosis by ground operations team due to the time for the spacecraft to communicate with the ground stations, on-board FDIR helps to respond to faults in a timely manner and increase the operational time of the spacecraft. Fig. 2.1 shows the role of FDIR from a malfunction to stable operation of the system. According to [WF13], FDIR generally comprises of the following procedure:

- Detection of the time and occurrence of a fault

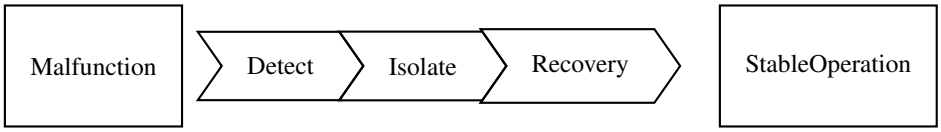


Figure 2.1: Fault Detection, Isolation and Recovery

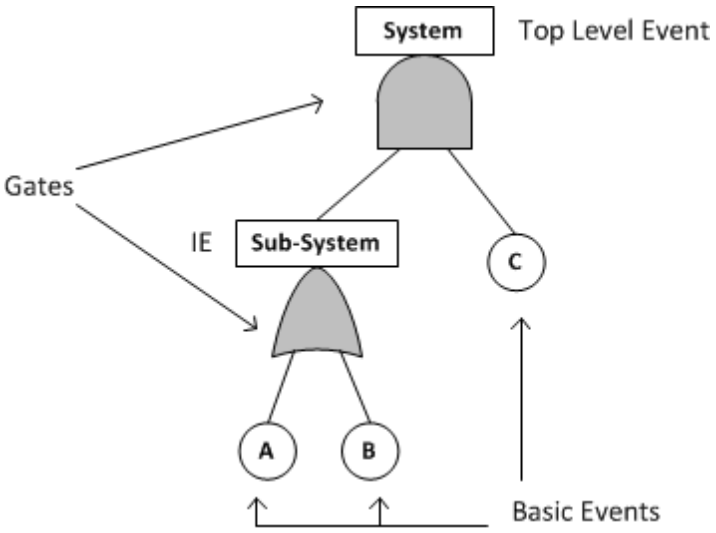


Figure 2.2: Fault Tree

- Isolation of the responsible component or subsystem and analysis of the fault
- Recovery of the system by reconfiguring it to a stable working condition

Various state of the art risk analysis techniques is used during design time to derive on-board actions in the event of a fault. One of techniques used in the space industry, which we focus on in this thesis, is Fault Tree Analysis.

2.2 Fault Tree Analysis

Fault Tree Analysis (FTA) is a state-of-the-art verification method used in many industries to perform failure analysis and check system dependability. It helps identify all possible error combinations that can lead to system level failure. They were first developed at Bell Labs and eventually used by Boeing for the safety evaluation for their commercial aircrafts [Eri99]. Since then, it has been adopted for different industries such as the aerospace, nuclear, chemical, robotics and software. The definition of Fault Tree Analysis from [Kri06] is:

> "*a top-down (deductive) analysis, proceeding through successively more detailed (i.e. lower) levels of the design until the probability of occurrence of the top event (the feared event) can be predicted in the context of its environment and operation*"

The general role of using Fault Tree Analysis in system analysis is given below:

1. Visualize the combination of failures and how it propagates through a system.
2. Identify commonly occurred errors like human errors, software errors etc.

3. Obtain a deeper understanding of system vulnerabilities and identify areas of concern.

4. Help prevent future failure by improving system design.

5. Identify means to minimize the probability of occurrence of failure in the future.

6. Monitor performance of the system.

Fault Tree Analysis requires the construction a graphical diagram called the Fault Tree to represent the system. It provides a mapping of the interactions of the failure causes with the system. Fault Trees are the logical diagrams that are used to model potential faults of the system. The definition of a Fault Tree from the Fault Tree Handbook [VGRH81] is:

> "*a graphic model of the various parallel and sequential combinations of faults that will result in the occurrence of the predefined undesired event*"

Fig. 2.2 shows the general structure and elements of a Fault Tree. They are directed acyclic graphs and each node in a Fault Tree is either a gate or event. The elements of the Fault Tree are described as below:

1. Events represent an occurrence in the system and can be of the types: Basic Event, Intermediate Event(IE) and Top Level Event (TLE). A basic event represents the component level failure and have a fail rate associated with them. They can also have a repair rate associated with them if the fail event is non-permanent. The basic events are represented as the leaves in a Fault Tree diagram. IE represent the sub-system failure i.e. the failure caused by two or more basic events. The TLE is the system level failure and is represented by the root node in a Fault Tree diagram.

2. Gates are logical elements used to model the interrelation between faults. It represents how the failure of the basic event propagates through the system and affect the top level event. The different types of gates are discussed in section 2.3.

2.2.1 Need for Semantics in Fault Trees

There are various techniques and methods to analyse the Fault Tree of a system. The techniques can be either qualitative or quantitative analysis. Qualitative analysis provides information about the vulnerabilities of a system from the structure of a Fault Tree [SD13]. However, quantitative analysis from the Fault Tree is only possible in some variants of Fault Tree. For most cases they need to be transformed to mathematical models for further analysis. Fig. 2.3 shows the transformations from a Fault Tree to derive system analysis. For the purpose of the transformation to the mathematical models from a Fault Tree, the semantics of the different versions of Fault Tree models need to be defined. It describes how the elements of a Fault Tree can be represented in a mathematical model with equivalent behaviour.

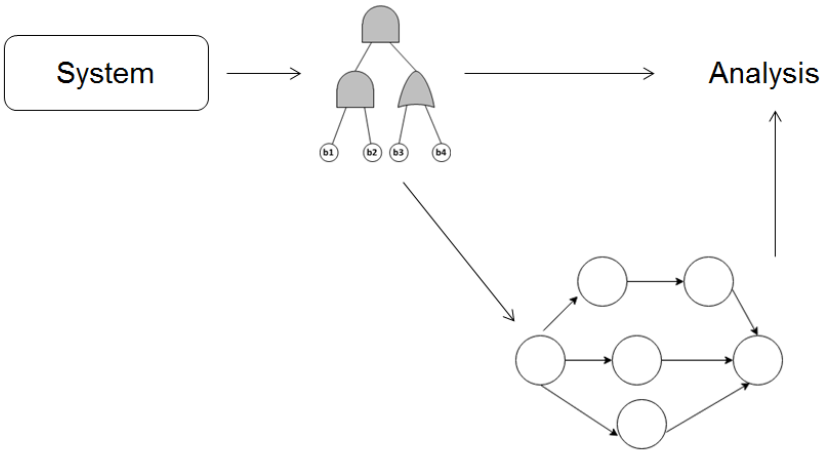


Figure 2.3: System analysis from Fault Tree

2.3 Types of Fault Trees

Over time, extensive research on the area of Fault Tree analysis have led to the development of different versions of Fault Trees. Static Fault Trees (SFT) are the earliest and most basic version of Fault Tree. All Fault Trees have evolved from these Static Fault Trees. Some of these versions which have been widely adopted in industries have been discussed in the following sections.

2.3.1 Static Fault Trees

Static (or Standard) Fault Trees (SFT) refer to the earliest version [RS15] of Fault Trees used in Fault Tree Analysis. These are the classical Fault Trees that were initially used for safety analysis. The Fig. 2.4 shows the gates that were used for SFTs, that are referred to as static gates. Static gates represent the static fault behaviour of the system, that is failure combinations. We discuss the different gates of a SFT below:

- AND Gate: It models system failure that is caused by combination of failure. For the gate to fail all the events connected to the AND input needs to fail.
- OR: It models system level failure caused by any one of its components. When any of the basic events connected to the OR gate occurs, the gate fails.
- Vote: k-Vote Gate can model failures when k out of N of its components need to fail for the gate to fail. Implementation wise both AND and OR gate can be modeled using the k-Vote gate using the N/N-Vote and 1/N-Vote gate respectively.

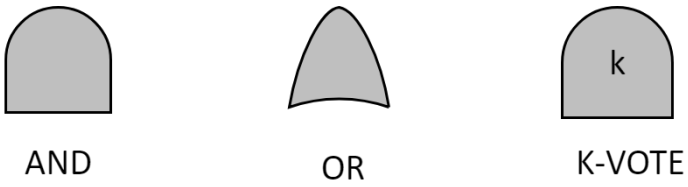


Figure 2.4: Gates used in a Static Fault Tree

In addition to the above gates, NOT gates are used in some cases to represent non-coherent systems which may include a component whose failure can repair the system or its functioning can cause system failure. They are generally dismissed as a modeling error [VGRH81].

The Fig. 2.5 shows a construction of a Static Fault Tree that uses the above described gates.

2.3.1.1 Analysis of SFTs

SFTs can be analysed qualitatively to get an idea of the vulnerabilities of the system. It involves looking at the failure combinations. The common methods used for SFT analysis are:

1. **Minimal Cut Sets (MCS)**: A Cut set is the set of components whose failure can cause a system to fail. Minimal Cut Sets contain the minimum number of components required to cause top level failure. A System is considered vulnerable if the minimum cut sets have very few elements or if they have elements with high likelihood of failure. Two of the most common methods to derive the cut sets of a system are based on: Boolean Manipulation and Binary Decision Diagrams [BA78].

2. **Common Cause Failure**: We can also analyse the Fault Tree qualitatively by enlisting the common cause failures of the system. Common Cause failures are failures of separate components that can be caused due to a common cause that cannot be modeled in the Fault Tree. In the space industry, CCF is defined by NASA PRA guide, 2002 as "*The failure (or unavailable state) of more than one component due to a shared cause during the system mission*". Examples of Common Causes are manufacturing defect, high temperature, lack of maintenance etc. Reliability of the system can thus be improved by identifying these common cause failures and taking appropriate steps like changing designs, avoid human errors, regular testing and maintenance etc., thus reducing the likelihood of the occurrence of common cause failures.

The SFTs can also be quantitatively to find numeric values for commonly used system metrics. The BEs are equipped with failure probabilities which are used for the computation of the metrics.

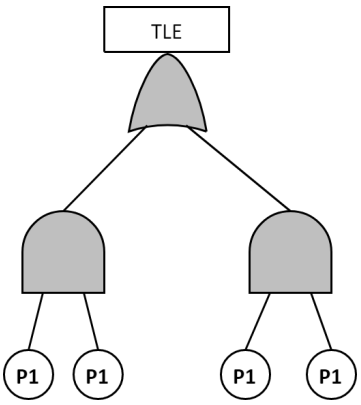
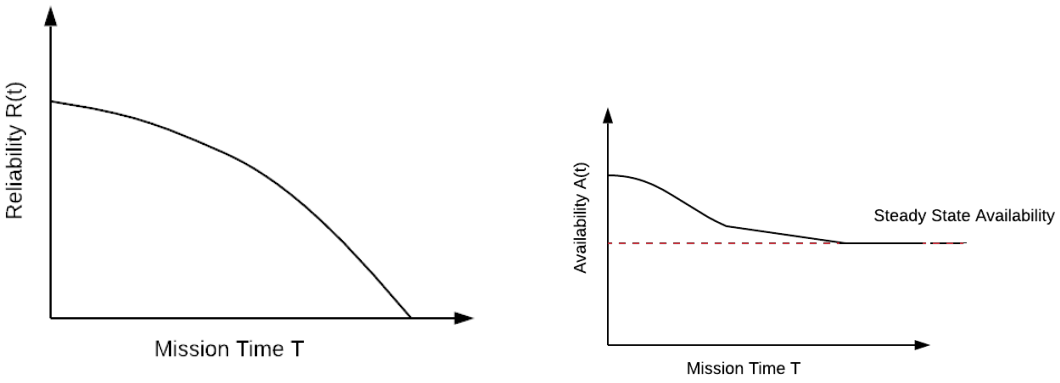


Figure 2.5: A Static Fault Tree

2.3.2 Metrics



(a) Reliability (b) Availability

Figure 2.6: Reliability vs Availability Curve

Fault Tree Analysis can measure the fault of the system qualitatively and quantitatively. We can qualitatively analyse simple fault trees by identifying cut sets as discussed in 2.3.1.1 and gain basic information about the risks and vulnerability of the system. However, to analyse more complex systems and to get information of the system vulnerability at different instances of time we perform quantitative analysis to get different metrics. Some of the commonly used metrics [Rel] [Ava] are:

1. **Reliability**: It indicates probability of the system functioning without any failure up to the time of measurement under given conditions. In system analysis, it is a commonly measured metric for non-repairable systems.

2. **Availability**: Availability is the probability of the system running at the measured time if it had run without any failure up to the time or if it had failed and repaired and runs

successfully up to the measured time, under given conditions. This metric takes into account that the system can be operational again after failure due to repair or transient faults. This metric is thus more relevant for systems with repairable components.

3. **Steady-State Availability**: It is the availability of the system as time tends to infinity. It gives an idea of the long run availability of the system. For non-repairable systems steady state availability is always zero as the system is expected to fail at some point in infinity and not become operational again.

4. **Mean Time To Failure**: It is the mean time for a system to operate without faults before the failure of the system. For a component with failure rate λ, mean time to failure (MTTF) is equal to $1/\lambda$.

The Fig. 2.6 shows the availability and reliability curves. As seen in the figure, the reliability curve ultimately settles to zero indicating that the system will definitely fail some time in the future. The availability curve however reaches a steady state value which is referred as the steady state availability.

SFTs are a graphical, intuitive and easy to understand tool that can be used for simple systems. However, SFTs are not expressive enough to model more complex systems as it only takes into account the failure combinations and not the order or timing of the failure. In many practical systems today the temporal ordering of the faults is also taken into consideration for determining the failure status. SFTs also have no support to model functional dependencies depicted by the components. For instance, a functioning CPU or a sensor has a functional dependency on the power. This has led to various extensions to the formalism to increase the expressiveness of fault trees.

2.3.3 Dynamic Fault Trees

Dynamic Fault Trees (DFT) are an extension to the Static Fault Tree. They can model temporal dependencies, functional dependencies and management of spare resources in addition to the existing semantics of a SFT. Temporal dependencies refer to the situation where the order of failure of the components in a system is important to a system. The system fails only when the components fail in a particular order when it has temporal dependencies. Functional dependencies in a system is when some of the components require another component to be functioning for its to be operational. Spare Management refers to the tasks involved in organizing the usage of spare or backup resources when the primary unit of the system component fails.

To implement the added expressiveness, the DFTs use the Priority-AND, Priority-OR, SPARE gate and Functional Dependency (FDEP) Gates in addition to the already existing static gates. Fig. 2.7 shows the gates introduced in Dynamic Fault Trees.

The different gates introduced by the DFT are discussed below:

1. Priority-AND gate (PAND) is an AND gate which fails only if the inputs fail from left to right. In some cases, the PAND gates can be inclusive ($\leq$) when the gate also fails when all the inputs fail simultaneously. Exclusive gates ($>$), on the other hand, allow strict ordering by only failing when the inputs fail left to right. The gates become fail-safe when the order is not maintained.

2. Priority-OR gate (POR) is an OR gate which fails only when the first input fails before the others. Like the PAND gate, they can also be both inclusive ($\leq$) and exclusive ($<$) and become fail-safe on not following the order.

3. Spare gates (SPARE) are used to model spare components in a system. The gate initially uses only the primary component which is connected as the first input. The spare components at this point can be in active or dormant mode. Components in dormant mode fail with a reduced failure rate determined by the dormancy factor d which lies in the interval [0, 1]. Spares with dormancy factor 0 are called cold spares and those with dormancy factor < 1 are called warm spares. Spares in active mode have a dormancy factor of 1. Upon failure of the primary component, the SPARE switches to the next spare component from left to right. The SPARE fails when there are no more spare components for it to claim.

4. Functional Dependency (FDEP) gates are used to model functional dependencies in a system. The first child is called the trigger and the rest of the connections are called dependent events. Unlike other gates they have no parents. Upon occurrence of the trigger event the FDEP forces failure of the dependent events.

The Fig. 2.8 illustrates an example of a Dynamic Fault Tree. It shows the use of the Spare gate, priority gate and FDEP gate by modeling a pump system. The functional dependency between power supply PS and motors P1 and P2 is shown. The temporal dependency between the switch S and pump P1 is depicted using the priority gates where the PAND gate fails only if the switch fails before pump. Since, when a switch has already failed, on failure of the primary there is no way for the system to switch to pump P2 with faulty switch. SPARE gate models the use of spare management with a back-up pump.

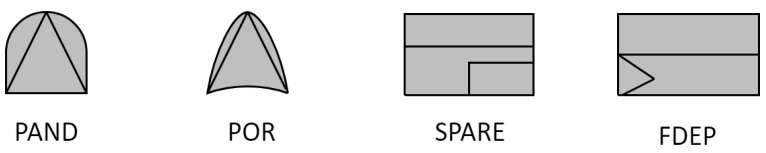


Figure 2.7: Dynamic Fault Tree Gates

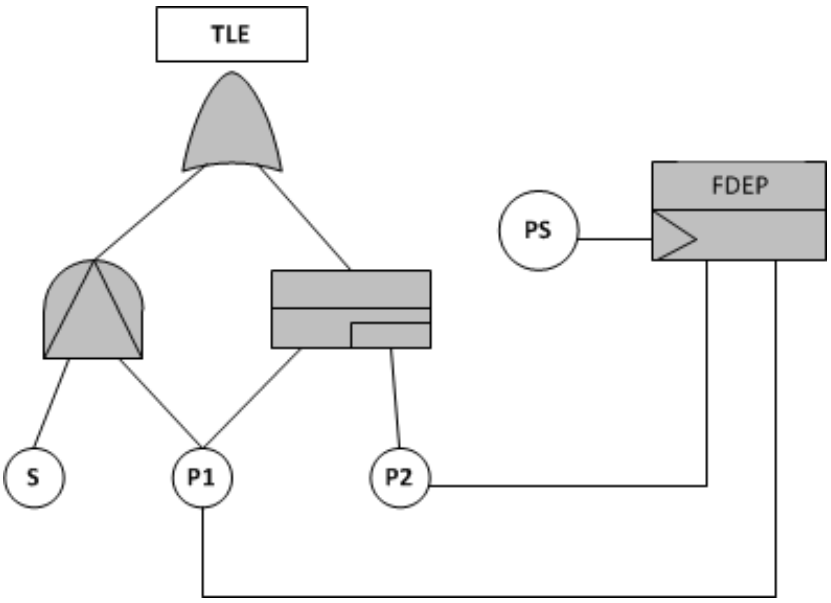


Figure 2.8: A Dynamic Fault Tree

2.3.3.1 Analysis of DFTs

With increased expressiveness of DFTs, comes increased complexity in its analysis. Analysis of DFT is no longer simple and requires sophisticated algorithms and tools. Since the temporal ordering is important, minimal cut sets are no longer enough for quantitative analysis. The paper [TD04] introduces the notion of minimal cut sequences for DFTs which is based on the Minimal Cut Set generation. It involves using a Zero-Suppressed Binary Decision Diagram method to generate the sequences. Zero-Suppressed Binary Decision Diagrams are special types of Binary Decision Diagrams used for computational set representation and manipulation.

The quantitative analysis of DFT can be done by translating it to a Markov Model and applying quantitative analysis techniques on the Markov models to obtain system metrics. The Markov models can be generated using a monolithic or compositional Approach:

- The monolithic method which is based on the Galileo Approach [VJK16] [SDC99], involves iteratively generating the states starting from initial state. The states are generated by considering the basic events in the DFT as failed, propagate the failure through the DFT and create a new state based on the resulting failure status of the DFT.

- The compositional Approach involves transforming the components or subsystems into Markov chains and do the computation for the individual components and combine the results.

Alternatively, DFTs can also be transformed to other models such as Petri Nets, Bayesian Network etc. for analysis. In this thesis, we use Markov Automaton for analyzing. Quantitative Analysis is used to measure numerical values to assess the dependability of the system.

More complex DFTs also lead to the state space explosion problem where the number of states increase exponentially with the introduction of new events. However, a number of reduction approaches are available to effectively handle the state explosion problem.

2.3.4 Repairable Fault Trees

Repairable Fault Trees (RFT) refer are the Fault Trees that can additionally model component with repair. The Fault Trees discussed so far modeled permanent failures. Therefore, we need to account for restoration, replacing components and transient faults as not all faults are permanent and some faults can be restored. Occurrence of transient faults and repair significantly increase the availability of the system. To capture this, Repairable Fault Trees were introduced. In the simple repair rate model, the Static Fault Trees have an additional repair rate associated with it which is a parameter for exponential distribution modeling the time to repair the component. It captures systems where the repair of the component is independent of other components and concerned only the repair of individual components. The Fig. 2.9 shows a Repairable Fault Tree with a repair box.

In the work of Bobbio [RFIV04] , repair boxes were introduced to extend the expressiveness of simple repair rate model. The repair boxes were connected to the gates and repairs the inputs to the gate when the gate fail. The repair boxes could carry different repair policies such as restriction on the number of components repaired at a time and different repair strategies, global or local repair or assigning priority to the repairs. This formalism was further developed to allow non-determinism to the repair policies in the work of [BCRFH08] and choosing the optimal repair strategy. The optimal repair policies are computed based on the costs for unavailability, failure and repair.

Repair with Dynamic Fault Trees have also been researched upon in some papers. In [MCD+12], a Repairable Dynamic Fault Tree (RDFT) was introduced that analyses the RDFT by converting to an ATS model. It maintains the strict order of claiming (left to right) for the SPARE gates and if a preceding component of the current active component gets repaired, the SPARE gate switches to the restored component, thereby maintaining the priority ordering. In case of Spare races, when two SPARE gates try to access a common shared spare resource, a random SPARE gate is allowed to claim the SPARE resource. For the FDEP Gate, the dependent events are set to an OFF state when a trigger event occurs, thus allowing them to fail or repair even in the off state.

Similarly, in [GKS+14] DFTs to capture repair and maintenance in the application of railways is introduced. The introduce work allows to compare different maintenance strategies using stochastic model checking techniques. Repair Modules (RMs) that monitor the components for failure are used and on failure sends a repair request to another entity called the Repair Unit (RU) which in turn is responsible for choosing the repair strategy and acti-

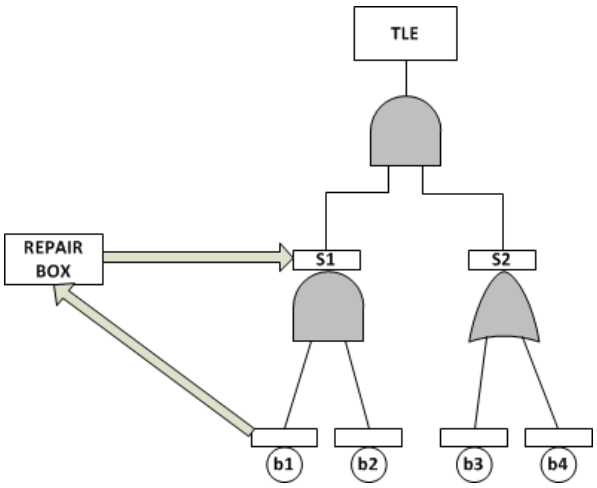


Figure 2.9: A Classical Repairable Fault Tree

vation of repair process on the component and sending repaired signal back to the RM on successful completion. The failed component is then reactivated by the RM. The RU also decides the sequence of fault repair in case of multiple failures, to optimize the system objective. [GKS+14] transformed the Fault Tree into Markov Chains. [GKS+14] analyses the Repairable Dynamic Fault Tree by converting it into Interactive I/O Markov Chains. In [RFIV04], the Analysis of the Repair box elements and the sub trees of the FTs affected by the repair were done by converting it into General Stochastic Petri Nets (GSPNs). This allowed evaluation of the repair policies.

[AHMS19] Introduces the concept of temporal Dynamic Fault trees to capture the effect of transient faults in a system. The Fault Tree formalism also introduces a Fault Tree Analysis methodology based on statistical model checking. The Paper defines the semantics for all static and dynamic gates except the SPARE gates.

2.3.5 Other Fault Tree Extensions

There have been other extensions to the Fault Trees introduced as well in the literature. State-Event Fault trees introduced in [KGF07] combines the elements of Fault Trees and State-machines to offer a more intuitive representation that can be used in industries. It is an extension to Component Fault Trees, which uses a component-based structuring for Fault Trees, by offering a graphical distinction between states and events. Unlike the Fault Trees, the Gates in this variant are not just Boolean logical functions but it encompasses internal state charts that capture the state changes in response to events. State Event Fault Trees are analyzed by converting to extended Deterministic Stochastic Petri Nets (eDSPN).

Fuzzy Fault Trees [TFLT83] includes the notion of Fuzzy sets to represent the failure probabilities instead of using exact numbers. It can effectively handle uncertainties in determining the failure probabilities of the components. Failure probabilities for system may be

difficult to estimate for systems with environment changes or for those components that have not failed yet.

Fault Trees with dependent events were proposed in [B$^+$99] that added dependencies between components such that the fail and repair rates were affected. Since in a realistic system, faults of one component can accelerate fault of another component. It allows modeling of gradual degradation of system by having additional states for it by allowing components to have more than failed and non failed state.

2.4 Existing Work

This section describes the existing work of Fault Tree on which the thesis is based. This was developed at the German Aerospace Center and the following section describes the

2.4.1 Non-Deterministic Dynamic Fault Trees

The Non-Deterministic Dynamic Fault Trees [MGN18] is an extension to Dynamic Fault Tree which introduces the concept of non/deterministic system recovery actions. The thesis work is based on this version of Fault Tree. The recovery actions related to the actions taken during spare management. It removes the syntactical restriction of fixed order claiming of spare resources for increased flexibility and achieve improved reliability. Non-deterministic recovery behavior also takes care of the problem of spare races which is not addressed in DFTs. Spare race is a condition when the primary resource of two spare gates sharing a common backup resource fails at the same time and both the spare gates try to claim the spare resource at the same time. DFT semantics do not specify the action to be taken in such situations. However, in the non-deterministic semantics the spare gates can choose to either not take any action or claim any of the available resources. The formalism defines the semantics of a spare gate to perform one of the recovery actions. According to the paper, the recovery action is formally defined as follows.

Definition 1. (recovery action): A recovery action r in an NDDFT T is an action of the form:

- [] (empty action) or
- CLAIM(G,S),such that spare Gate G claims spare S,where S is a spare of G

The formalism does the analysis of the Fault Tree by converting it into a Markov Automaton. It uses the monolithic approach to generate the state space of the Fault Tree. Two sets of data are memorized: the history of occurred basic event sets and a mapping of the Spare Gate to the claimed spares. Starting with the initial state, all active basic events are used to determine the successor transitions and states. The process is iterated for each generated

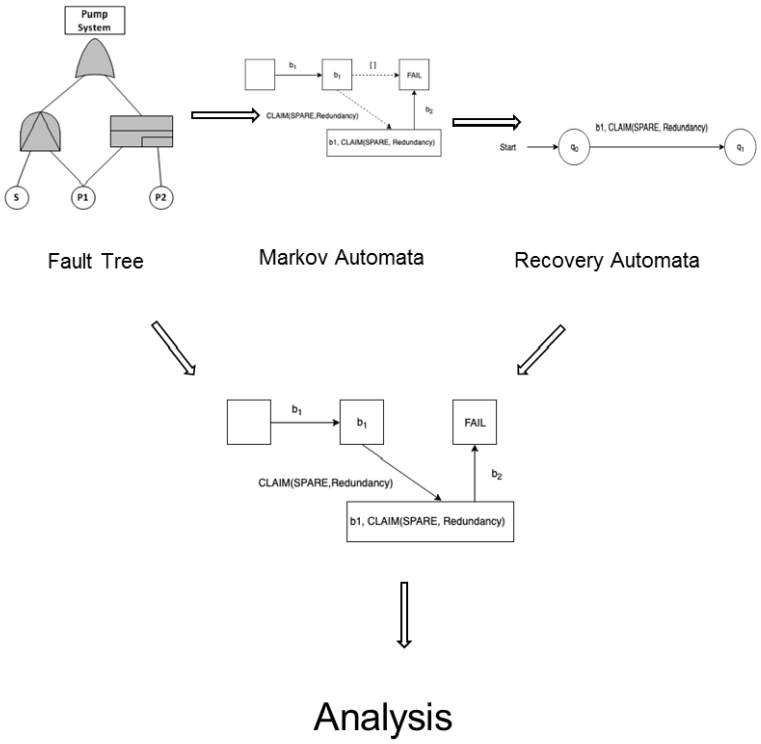


Figure 2.10: Workflow of Fault Tree Analysis using NDDFT

state and possible events in the state till there are no new states generated. All fail states are represented as a single final "FAIL" state.

Fig. 2.10 represents this transformation process in an NDDFT. The core elements of the system are described in the following section.

2.4.1.1 Markov Automaton

Markov Automata (MA) are finite transition systems used to model systems with non-determinism. It includes both immediate action based non deterministic transitions and Markov Transitions which execute with an exponential time delay. Probabilistic Transitions have probabilistic distribution for successor states. The Fig. 2.11 illustrates a simple Markov Automata. In the figure, the non-deterministic transitions are depicted using dotted lines, the markovian transitions depicted by non-dotted lines and the final states are represented by double circles. The NDDFT on which the thesis work is based on uses a Markov Automata for its Fault Tree Analysis. The extension work in the thesis also continues to use the Markov Automata as its a good model for representing both the non-deterministic and the timed markovian transitions. Additionally, the Markov models have been successfully used in a number of space ground segments due to its capability to produce good approximations for steady state availability of some space system (ECSS-Q-ST-30-0931). The formal definition of Markov Automaton based on [EHZ10] is,

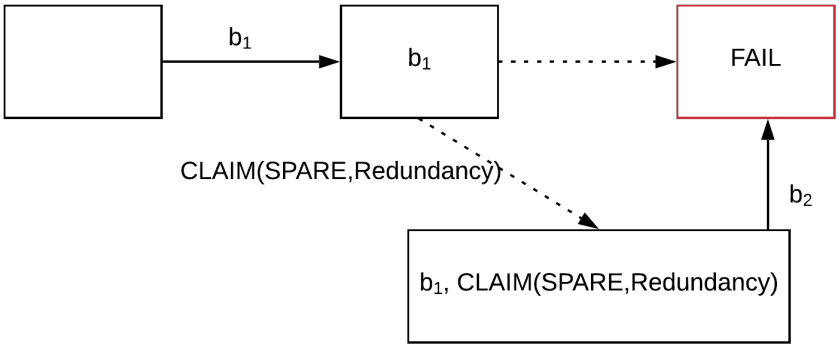


Figure 2.11: Markov Automata

Definition 2. (Markov Automaton) Markov Automaton is a 5-tuple $(S, Act, \rightarrow, \Rightarrow, s_0)$ where,

- S is a non empty finite set of states,
- Act is a set of actions,
- $\rightarrow \subseteq S \times Act \times Dist(S)$ is a set of Probabilistic transitions,
- $\Rightarrow \subseteq S \times R_{\geq 0} \times S$ is a set of Markov Timed transitions and
- $s_0 \in S$ is the initial state.

The non-deterministic transitions are chosen by a scheduler. Depending on the type chosen the schedulers then calculate the failure probabilities for the non-deterministic transitions from the given state.

Using existing Markov analysis techniques, a number of metrics can be calculated such as Reachability, long run rewards etc.

2.4.1.2 Recovery Automaton

The next step in the work-flow of NDDFT is synthesizing a scheduler for the Markov Automaton. The value iteration algorithm is used to find the most optimal path for maximum system reliability and produce a recovery automaton. A recovery automaton is a deterministic automaton that represents the best transitions that the Markov Automaton should take when faced with non-determinism. The definition of Recovery Automaton is given as:

Definition 3. (Recovery Automaton): A Recovery Automaton $R_T = (Q, \delta, q^0)$ of an NDDFT T is an automaton in which

- Q is a finite set of states,
- $q^0 \in Q$ is an initial state, and

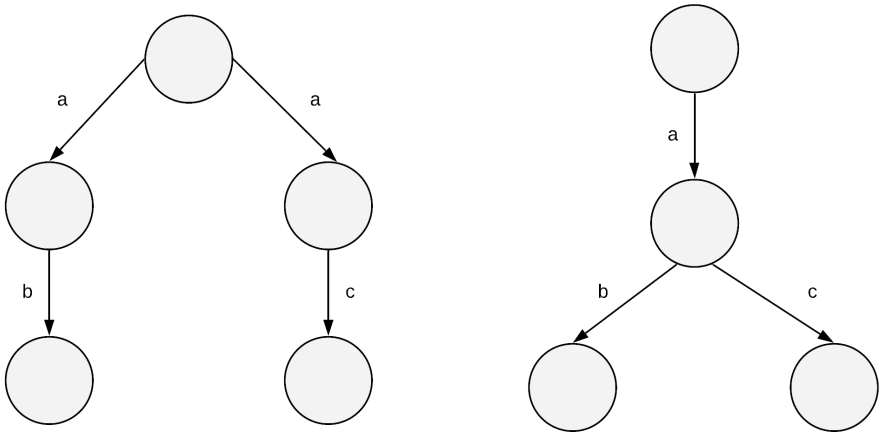


Figure 2.12: Two Labeled Transition Systems that are trace equivalent

- $\delta : Q \times BES(T) \rightarrow Q \times RS(T)$ is a deterministic transition function that maps the current state and an observed set of basic events to the successor state and a recovery action sequence.

Minimization in NDDFT However generated recovery automata may suffer from the problem of state space explosion. It is desirable to find a smaller automaton with equivalent recovery behavior. The properties or rules that were applied to reduce the state space of recovery automata are given below:

- **Trace Equivalence:**

  The notion of trace equivalence is used to merge equivalent states in systems where we can not influence the interact with the system. A trace of a state is the sequence of transitions or actions that can occur in the given state. Two states are said to be trace equivalent when the set of traces are the same. When two transition systems, have trace equivalent initial states, we consider them trace equivalent. For example, in the Fig. 2.12, both transition systems have trace set = ab, ac and are considered trace equivalent.

  In the minimization of NDDFT, the notion of trace equivalence is used with partition refinement algorithm to find trace equivalent states and produce a equivalent Recovery Automaton with reduced number of states and transitions.

- **Orthogonal States:** In the work of [MMGN18], recovery automata are further minimized by merging non trace equivalent states if they satisfy the property of orthogonality. This algorithm works on the assumption that every basic event can occur at most once in a non-repairable system.

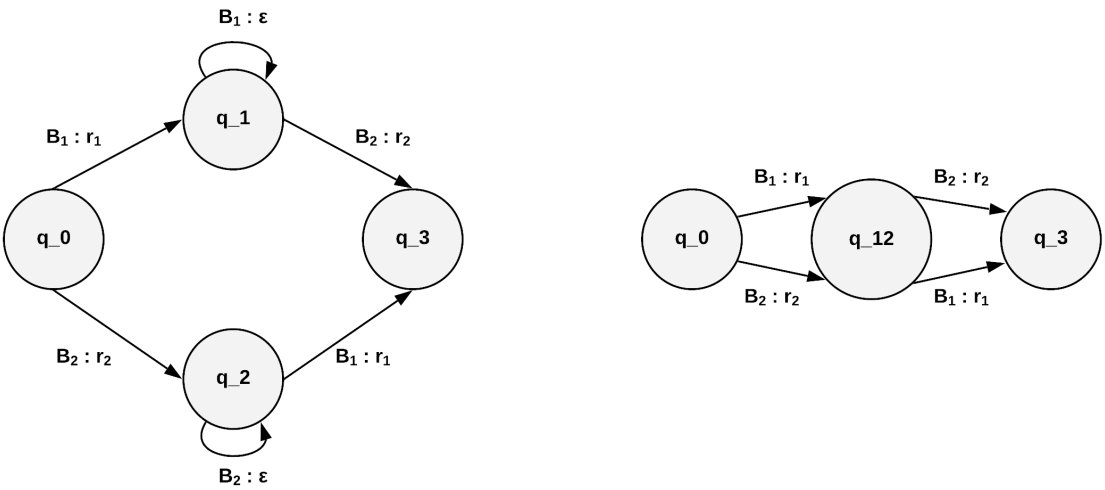


Figure 2.13: A Recovery Automaton a) before and b) after applying the orthogonal state rule

To capture the basic event sets that can no longer be produced upon reaching a state, this rule calculates the guaranteed inputs for each state. Using the calculated inputs, the rule eliminates transitions that can no longer occur and merge the orthogonal states. The Fig. 2.13 depicts the application of this rule for a small automaton. The states q_1 and q_2 are orthogonal with respect to B_1 and B_2 and thus can be merged to the equivalent state q_{12} directing all the incoming and outgoing transitions of q_1 and q_2 to and from this new merged state. The loop transitions at q_1 and q_2 are also eliminated as they can no longer occur.

Merging Final States: In this rule, the FAIL states that have no recovery transitions are identified. The transitions of the state that only lead to the FAIL state are turned into a self loop and the Fail state removed in case its unreachable. The definition of a FAIL state taken from [MMGN18] is:

Definition 4. (FAIL State). Let $R_T = (Q, \delta, q_0)$ be an RA and $q \in Q$ a state. Then q is a FAIL state iff for any $B \in BES(T)$, all transitions from q are of the form $\delta(q, B) = (q, \varepsilon)$.

The Fig. 2.14 shows the abstract representation of the application of the rule. Using this rule, the FAIL state q_2 is eliminated and the transition from q_1 to q_2 is turned into a self-loop.

2.4.1.3 Markov Chain

Markov Chain is used to represent the Fault Tree after the non-determinism has been solved from the information derived from the recovery automata. It is a labeled transition system

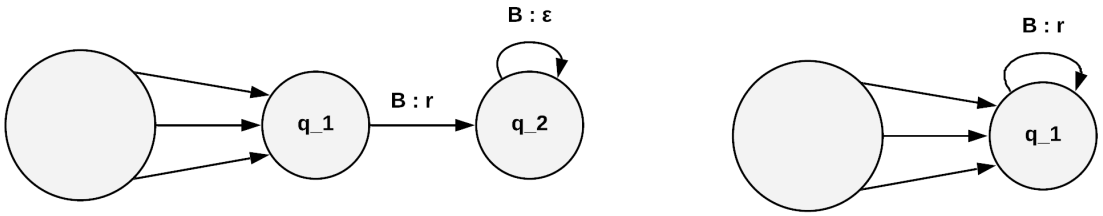


Figure 2.14: A Recovery Automaton a) before and b) after applying the final state rule

that represents the sequence of events with each transition having a probability distribution. Fig. 2.15 shows the resulting Markov chain representation of the Fig. 2.11 that was generated with the help of scheduler information from Recovery Automaton.

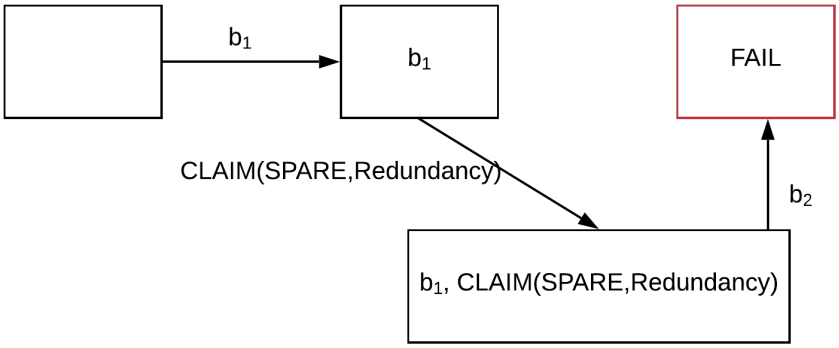


Figure 2.15: A Markov Chain

2.5 Fault Tree Analysis Tools

Active Research in the field of model checking has led to a number of Fault Tree analysis tools available. A brief discussion of some of the popular tools used in the industry is given below.

2.5.1 Galileo

Galileo [SDC99] is a popular software tool for analysis of Dynamic Fault Trees developed at the University of Virginia . It solves Fault Trees using a combination of Markov models and Binary Decision Diagrams. It introduced the use of Galileo specification for Fault Trees which has been adapted and used widely for specifying the Fault Trees in a textual manner.

2.5.2 DFTCalc

DFTCalc [ABVdB+13] is a verification Tool developed at University of Twente in the Formal Methods and Tools group. It is used for DFT analysis and utilizes compositional semantics for modeling. The elements of the DFTS are modeled as IO-Interactive Markov Chains. The DFT in Galileo form and the mission time is given as the input to the tool which can then calculate the unreliability of the system. It also includes support to measure Mean Time To Failure and the steady-state availability. It comes with a web tool with user friendly UI and it utilizes a number of commonly used model checkers to provide the tool.

2.5.3 PRISM

PRISM [KNP11] is an open source probabilistic model checking tool, implemented in C++ and Java. It can analyze different probabilistic models: Markov Chains, Markov Decision Processes, Probabilistic Automaton and Probabilistic Timed automaton with the option to include cost or reward to the models. It also has support for statistical model checking and model checking stochastic multi-player games. It offers a Graphical User Interface (GUI) as well as a Command Line Interface for the users. The input is given in the PRISM language representation. It includes different algorithms and data structures like the BDD and multi-terminal Binary Decision Diagrams. However, it does not have support for Markov Automaton.

2.5.4 MRMC

The Model Reward Model Checker (MRMC) [KZH+11], developed in C, is a tool developed for model checking probabilistic models. It supports continuous and discrete time Markov models: Discrete-time Markov Chains, Continuous-Time Markov Chains, Discrete Time Markov Reward Models, Continuous Time Markov Reward Models and Continuous Time Markov Decision Processes. It provides a command-line tool and the input is given in the form of files: a .tra file specifying transition rates, .lab file with the labels for each state, a .ctmdpi file with the rate matrix and transition labeling of a CTMDPI and optionally .rew and .rewi files. The .rew file specifies the reward structure and .rewi specifies the reward impulse structure.

2.5.5 STORM

Storm [DJKV17] is a probabilistic model checking tool, implemented in C++, developed at RWTH Aachen which is open source since 2017. It provides model checking support for Markov Chains, Markov Decision Processes and Markov Automaton well as extensions with

reward structure. The tool is flexible with the input model as it can parse a wide range of model input formats such as PRISM, JANI, explicit etc. It features in memory representation of the models as either sparse matrices for smaller sized models or multi-terminal Binary Decision Diagrams for larger models. It includes the support of 15 different solvers in the back end which are used by the model checking engines for different tasks. However, it does not provide support for LTL model checking, statistical model checking and probabilistic timed automaton. It offers interface via C++ API, python API and a command line interface for users.

Chapter 3

Concept

This chapter explains the conceptual aspects of the contribution in this thesis. It explains how the proposed semantics extends the Non-Deterministic Dynamic Fault Tree formalism, the extended Recovery Automaton model and adapted algorithms for the minimization of Recovery Automata.

3.1 Non-Deterministic Repairable Fault Trees

Non-Deterministic Repairable fault trees are extensions of the Non-Deterministic Dynamic fault trees with support for repair of components. It combines the semantic flavour of dynamic fault trees, non-deterministic recovery behavior and repair capability of the components. The repair event moves the failed component from a failed state to operational state. As is in the case of fail event, the repair too propagates upwards towards its parent nodes. Having repairable components in a system can affect the system in the following ways:

- Increase the availability of the system.
- Fail states may no longer be absorbing.
- A basic event can occur more than once if its repairable.

The Fig. 3.1 shows the abstract representation of the hierarchical relationship between the different fault trees. The figure shows how the different variants of fault tree are extending the previous version of fault tree with added functionalities. To support the modeling of repairable systems as a non-deterministic repairable fault tree the gate semantics and the associated definitions and algorithms of the NDDFT needs to be extended. The following sections describe the relevant proposed modifications.

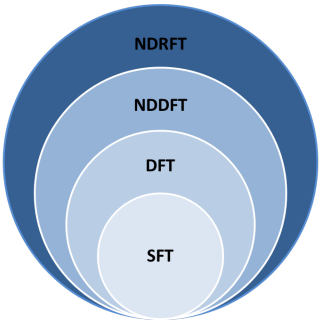


Figure 3.1: Relationship between different Fault Trees

3.2 Semantics

The semantics of the different gates in the non-deterministic fault trees need to be defined since quantitative analysis of fault trees with dynamic gates is not possible directly from the fault tree. The semantics is needed to transform the different elements in the fault tree into an equivalent Markov model representation. In literature, semantics for Non-Deterministic Dynamic Fault Trees[MGN18] have been defined as well as some work on Dynamic Repairable Fault Trees [MCD+12], but there is no work in literature that combines the advantage of non-deterministic recovery behavior of NDDFT with the consideration of repair. For this reason, it is important to define the semantics of NDRFT so that we can model and analyze repairable systems with flexible recovery actions.

Before defining the semantics, the notations and sets used that are used need to be stated:

- The fault events as stated in the previous work of NDDFT are denoted as $b_1^r, b_2 \ldots b_n$ and sets of fault events as $B_1, \ldots B_n$. We denote the repair events as $b_1^r, \ldots, b_n^r$ and sets of repair events as $B_1^r, \ldots B_n^r$. We use the sets to account for a possibility that multiple components that are functionally dependent on a common component can fail or restart together with the component that they are dependent upon. The set of non-empty repair events is denoted as $RES(T)$.

- Since the events may be either repair or fault events, we denote a set of events as $E_1, \ldots, E_n$ and the set of all non empty set of events as $ES(T)$ where $ES(T) = BES(T) \cup RES(T)$.

- Every element in $RES(T)$ has its corresponding inverse event i.e. the fault event in $BES(T)$. However, not every element in $BES(T)$ does not have its equivalent repair event in $RES(T)$ as some may not be repairable.

Having stated the notations, let us now look into the semantics of each individual gate.

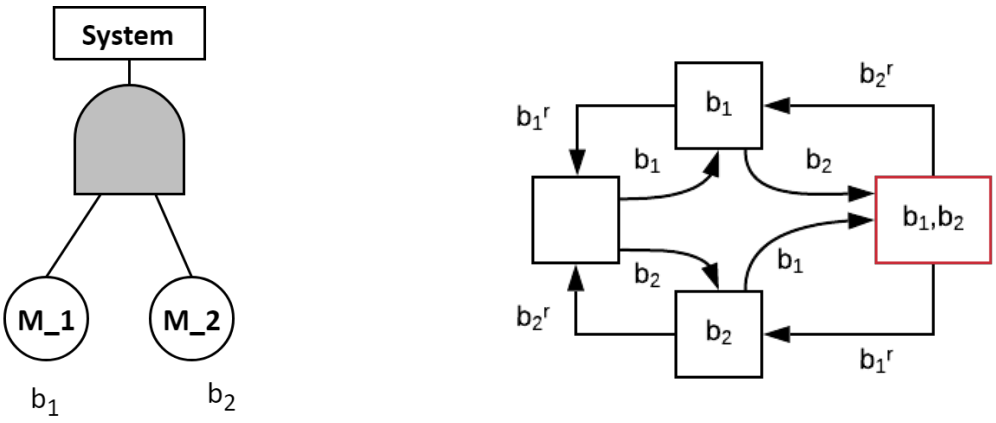


Figure 3.2: (a) AND Gate and (b) Markov Chain generated from AND

3.2.1 Semantics of Gates

Gate Semantics define how the fault tree is transformed to a Markov model for its analysis. This section discusses the proposed semantics for the gates in case of Repairable Systems.

3.2.1.1 AND

In the proposed semantics, the AND gate semantics for fault events remain the same. The gate fails when all of its inputs fail. In the case of the repair of any of its components, the gate moves from failed to a non failed state just as in a real system, on a shutdown of a system caused by a combination of failures, repairing any of the defective components can bring it back to a functioning state. The semantics is illustrated with an example.

Example 1. As an example, we consider the memory unit sub-system MU of a computer system. The memory unit has two redundant memory cards A and B. The combined failure of both cards cause failure of the sub-system MU. This can be modeled with an AND gate as shown in the Fig. 3.2. The resulting generated Markov chain shows that the gate moves into the Fail state when both its components have failed. The failure transitions of A and B are denoted as b_1 and b_2 respectively. We can also see that the repair transitions b_1^r and b_2^r change the state of the system from a failed state to non-failed state.

3.2.1.2 OR

In the proposed OR gate semantics, the fail states with different failed components need to be differentiated. In a non-repairable version, the fail states could have been merged as any further transition from the failed state won't change the fail status of the gate. However, with repair semantics a repair transition affects the gate's fail status. Repair of any of the failed

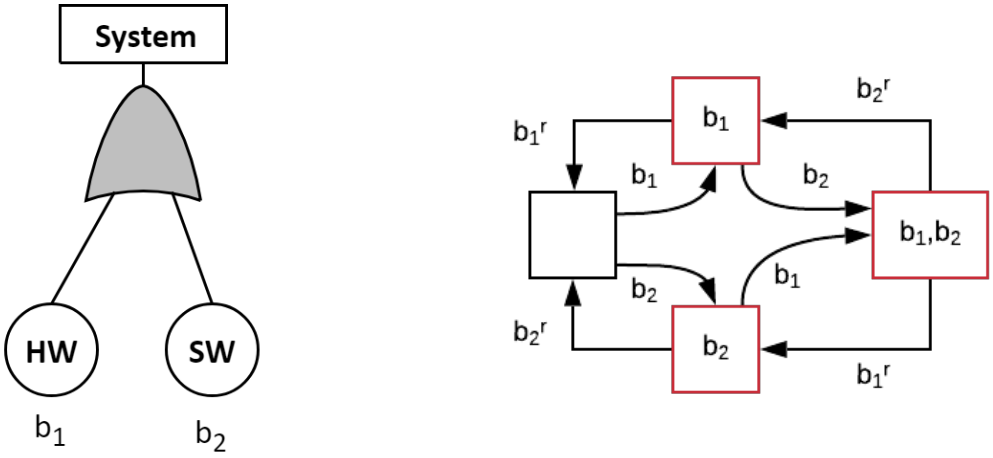


Figure 3.3: (a) OR Gate and (b) Markov Chain generated from OR

component moves the state to another failed state if any other component is still in failed state. The gate returns to its functioning state when all of its failed components have been restored. This illustrates the real behavior of a system which has components that relate to the top-level failure with an OR gate. The semantic is further illustrated below.

Example 2. Let us consider a computer system with two possible causes of system failure: A hardware error and a software error. Occurrence of any of the events can cause the system to fail. Restoration of the failed component brings it back to its previous state. This can be modeled by the OR gate as shown in Fig. 3.3. As shown in the corresponding Markov chain generated from the OR semantic, there are 3 fail states and one non fail state. The fail events of Hardware Error and Software Error are denoted as b_1 and b_2 respectively. For the system to return to its non failed state, both repair events need to have occurred.

3.2.1.3 PAND

The PAND semantics, like its non-repairable behavior, fails only if the order of failure is from left to right. Any other order is not considered a fault. From the fail state, repair of any of its components can bring it back to its non failed state as now the current failed components do not fulfill the basic property of PAND gate that requires all of its child inputs to have failed, for it to be in fail state. We understand the PAND semantics with a practical example.

Example 3. The example of a system with two pumps and a valve [JGKS16] is considered to illustrate the semantic of a PAND gate with repair. The pump A is the primary resource and pump B is the backup which can replace the primary unit after the valve opened the pipe to the second pump. If the valve fails followed by Pump A, the system reaches the fail state as we can no longer switch to the backup pump.Repair of either primary or valve can bring it back to functioning state. Repair of pump enables us to open the pipe to the backup pump,

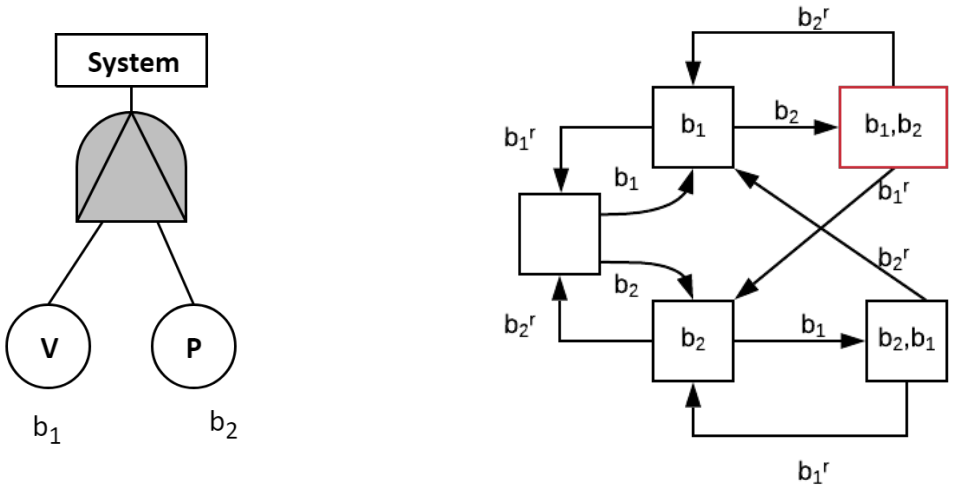


Figure 3.4: (a) PAND Gate and (b) Markov Chain generated from PAND

thus, bringing us to functioning state. Alternately, the repair of primary can also bring the system from failed to non failed state. We can observe from the generated Markov Automata in Fig. 3.4 that for fault sequence b_2, b_1 the system goes to a non-failed state. The system moves into the non-failed state only for the sequence b_1, b_2. It returns to non-failed state when either of the components repair event occurs.

3.2.1.4 POR

The POR gate fails as soon as the left most input fails before any of the other inputs have failed. Any other order of failure will cause it to move into a non failed state. Repair of the first child input brings the gate to a functioning state as it does not satisfy the basic property of a POR gate that requires the first input to stay in failed state. We discuss the POR behavior with a practical example.

Example 4. The behavior of POR gate is illustrated from the example of two computing devices and an actuator connected by data link [JGKS16]. The system has two redundant devices and is functional as long as one of it is in working state and no device blocks the data link by constantly sending messages into the data link blocking communication with other devices. Devices that block data communication are called babbling idiots. The dynamic fault tree representations of the example is shown in Fig. 3.5. The system fails if either the devices D fail or one of the devices turn into babbling idiots BI. The devices can fails if either the processor stops working or a data link failure occurs. The device becomes babbling idiot if the data link has failed before the failure of the processor.

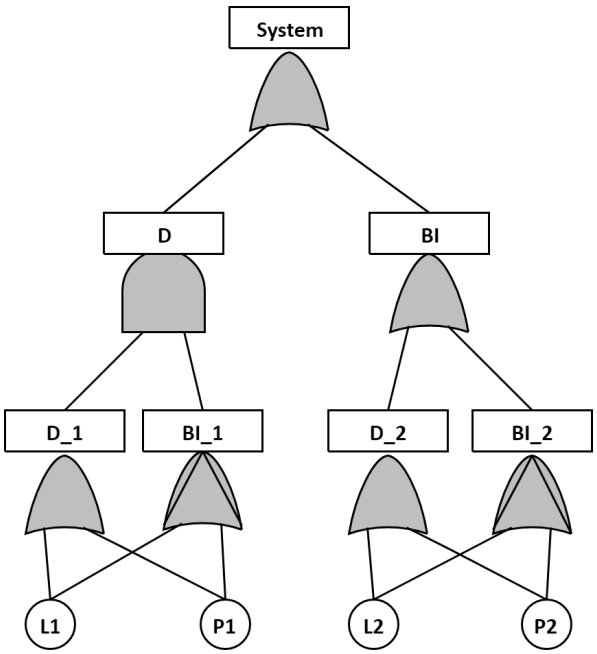


Figure 3.5: Example Computing Device and Actuator connected by Data link

This babbling idiot occurrence is modeled using a POR gate with the data link as left input and Processor as the right input. The figure shows the generated Markov automaton for the subsystem of a babbling idiot. When the link fails, the system moves into the failed state. Restoration of the link stops the device from being a babbling idiot and moves it back to the previous state.

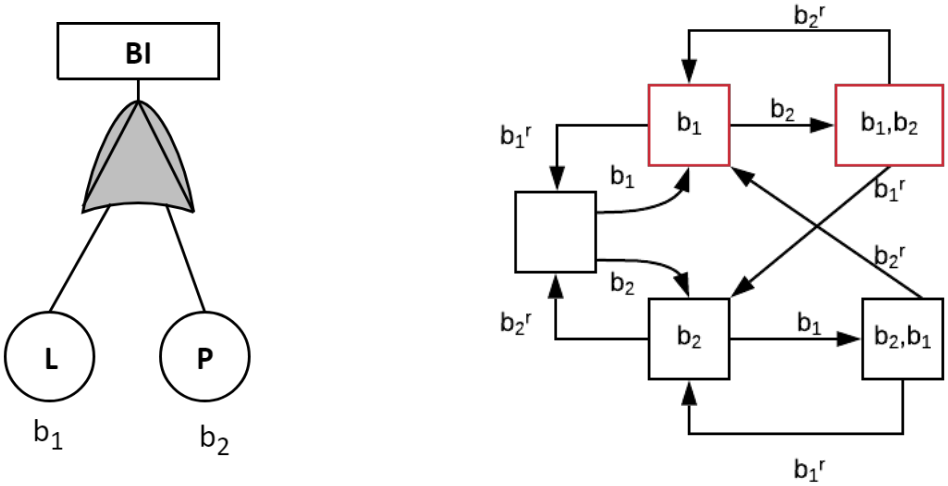


Figure 3.6: (a) POR Gate and (b) Markov Chain generated from POR

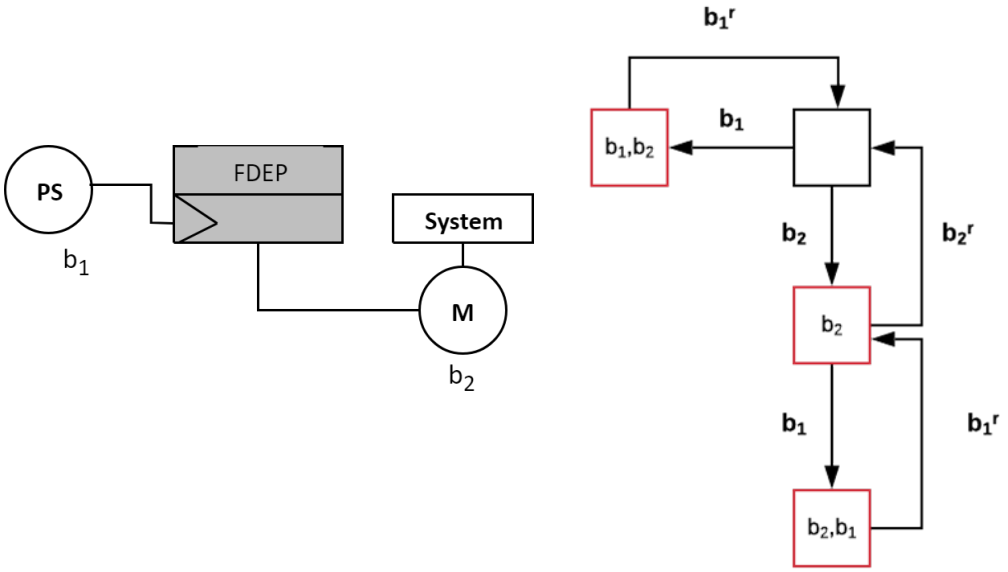


Figure 3.7: (a) FDEP Gate and (b) Markov Chain generated from FDEP for case 1

3.2.1.5 FDEP

The FDEP gate has a trigger input and dependent outputs. The occurrence of a trigger failure event propagates the failure to the functioning dependent events. If the trigger is repairable and it gets repaired, the FDEP propagates the repair to its dependent components that failed due to the FDEP trigger as well, thus bringing the dependent events to functioning state again. Dependent components that were already in the failed state before the occurrence of the FDEP's trigger event, remain unaffected by the fail and repair event of the FDEP trigger. We assume that the repair of the dependent events cannot occur as long as the component it is functionally dependent on is in failed state, since it would require the defective trigger event on which it has its functional dependency to be working to carry out further repair work.

Example 5. For illustrating the FDEP semantics, we consider a simple system that has a motor which is functionally dependent on the power supply P. The system is functional as long as the motor is functional. We consider both the motor and power supply are repairable. The generated markov chain based on FDEP semantics show that the power supply which is the trigger propagates its failure and repair to its dependent event Motor. If the motor was in a fail state before the failure of the power supply, the repair of the trigger does not propagate to the motor. The Fig. 3.7 shows the generated markov chain for the given case.

3.2.1.6 SPARE

The SPARE gate semantics is extended from the semantics of the Spare Gate in Non-Deterministic Dynamic Fault Tree. The SPARE gate is responsible for the non-deterministic transitions in the fault tree. In the NDDFT, the spare gate can switch to any of the available spares or not take any action when the primary fails. On failure of the claimed spare resource, the spare gate can again chose to claim any of the working spares or not take any action. However, in cases where the resources are repairable, the resource can come back to its functioning state and become available to the spare gate for claiming. When the Primary resource gets repaired, the Spare gate can either free its claimed spare resource to switch back to its primary or continue working with its currently claimed resource. The SPARE gate deactivates the claimed spare before and returns it back to its previous state before switching to its primary. The action to be taken by the SPARE gate is determined by a scheduler that is calculated from the generated Markov Automaton which is discussed in the following sections.

In the proposed concept, an additional action called the Free Action is introduced to allow the Spare gates to switch back to its primary. We require this action in an NDRFT as the primary resource might be repairable and become operational again when its repair occurs, and this action allows the flexibility for the system to go back to using its primary resource again after it has been repaired. These actions have been referred to as the recovery actions. It is the action taken by a system for management of its spares.

The revised recovery actions in an NDRFT can be formally defined below:

Definition 5. (Recovery Action): A Recovery Action r in an NDRFT T is an action of the form of one of the following:

- $[]$ (empty action) or
- CLAIM(G,S),such that spare Gate G claims spare S, where S is a spare of G or
- Free(S), where the claimed Spare S is freed

Thus in an NDRFT, the system can perform FREE action in addition to the already existing [ ] and CLAIM action.

As in the case of NDDFT, in the NDRFT semantics the actual recovery action r is defined by recovery strategy. $R(T)$ is the set of all recovery actions possible in an NDRFT T. The set of recovery action sequences $RS(T) = (R(T) \setminus \{[]\})^*$. Given the observed events (repair or fault), a recovery strategy is then a mapping that returns the recovery action sequence that should be taken accordingly. The NDRFT considers recovery strategies that only have recovery actions as defined in Definition 5 and is defined as follows:

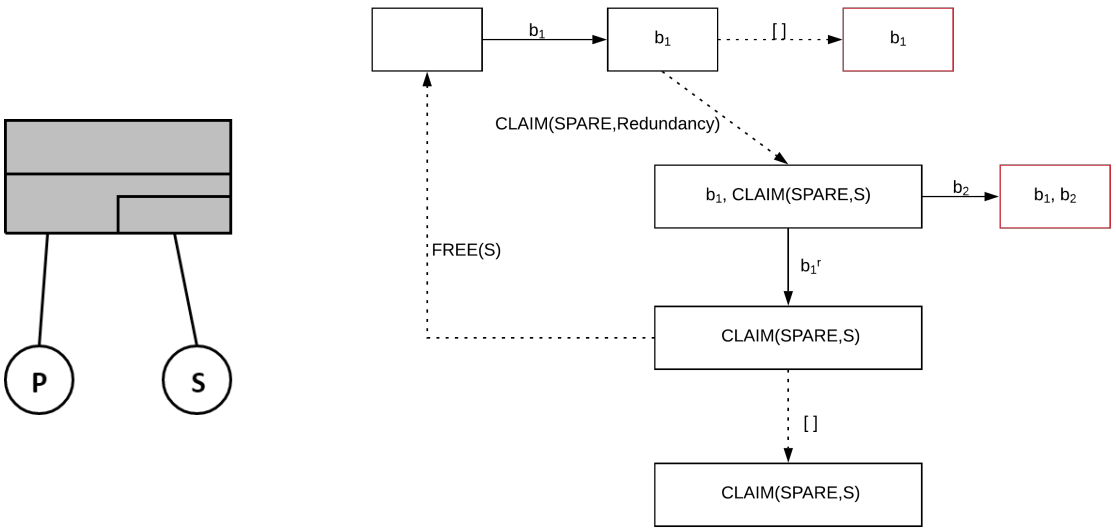


Figure 3.9: Spare Gate and section of generated Markov automata

Definition 6. (Recovery Strategy): A recovery strategy for an NDRFT T is a mapping $Recovery : ES(T)^* \rightarrow RS(T)^*$ such that

- $Recovery(\varepsilon) = \varepsilon$ and
- $Recovery(E_1, \ldots, E_n) = \text{Recovery}(E_1, \ldots, E_{n-1}), rs_n$ with $rs_n \in RS(T)$

We consider different examples to demonstrate the semantics for the different ways spare gates are used in a system:

- The first example demonstrates the simple use case of a spare gate which has one primary resource and one backup resource.

  Example 6. Let us consider an example of a Motor Unit in a system that consists of a primary motor P and a back up motor B that replaces it when failed. The fault tree construction of the setup uses a spare gate with a primary input to the motor P and a single secondary input to backup resource B. Fig. 3.9 shows the Fault Tree construction and a section the generated Markov Automata of the example. As seen in the diagram, after every fail action, the system has a choice to either claim the secondary resource or not take any action. as denoted by the dotted non-deterministic lines. Similarly, after the repair action, the system can choose to free its primary resource or not take any actions.

- This example demonstrates the use of two input SPARE and PAND gate combination which is commonly used in dynamic fault tree constructions. The PAND and spare share a common input which is the second input to the PAND and first input to the

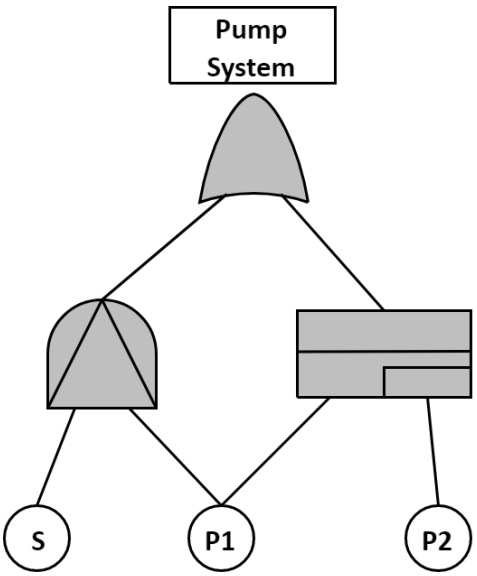


Figure 3.10: Pump System using PAND - Spare combination

spare. In the NDDFT, where the faults are permanent, failure of the first input to the PAND means that the Spare cannot take any recovery action when the Common input fails.

Example 7. Let us consider an example to understand this semantic. Consider the pump system whose dynamic fault tree is shown in Fig. 3.10. Failure of the valve before the pump, means that the pump system cannot switch to the backup pump when its primary fails. When the system becomes repairable, when the valve gets replaced after the primary fails, the system can switch to its backup resource as the valve becomes available and hence the system becomes operational. The valve input which is connected to the PAND gate, thus indirectly influences the recovery actions taken by the spare in the NDRFT semantics. In the repairable semantics, when the first input to the PAND is repaired after the gate has failed, if it shares the second input with a spare gate, it induces the non deterministic recovery transitions from the spare gate.

- The spare resources can often be shared by two gates in a dynamic fault tree construction. This represents the systems that have a common set of backup resources that gets shared by two are more primary components that are redundant in a system. Failure of any of the primary components, the failed component can choose to claim one of the available spares or not take any recovery action.

  Example 8. Let us consider a hypothetical system that has a CPU unit with two redundant processors and a pool of backup processors. The figure shows the dynamic fault tree construction of this scenario. According to the spare semantics, when either of the

primary processors fail, the system can use any of the available processors. The figure shows a section of the generated automaton and we can see that the

3.3 Transformation of Fault Tree to Markov Automata

As a next step i the workflow, the defined semantics of the gates are then used in the transformation of the fault tree to the Markov Automata for analysis. The transformation is done to a Markov Automata rather than any other Markov model as they allow the representation of non-deterministic actions in addition to the markovian transitions. The Markovian transitions here represent the basic events i.e. the repair or fault events. The non-deterministic transitions represent the recovery actions that can be taken, in this case, recovery actions related to spare resources.

The algorithm for transforming NDRFT to Markov Automaton is adapted from the Markov Automaton generation algorithm of NDDFT 2.4.1.2. It is originally based on [DBB92]. The adapted algorithm will use the updated DFT semantics that have been defined above for the transformation. The transitions generated can be either Markov transition or non-deterministic Transitions. The markovian transitions in an NDRFT can be either repair transition or failure transition. Additionally, each state will also store a list of currently failed components and generates the next state and transitions using this information and the DFT gate semantics.

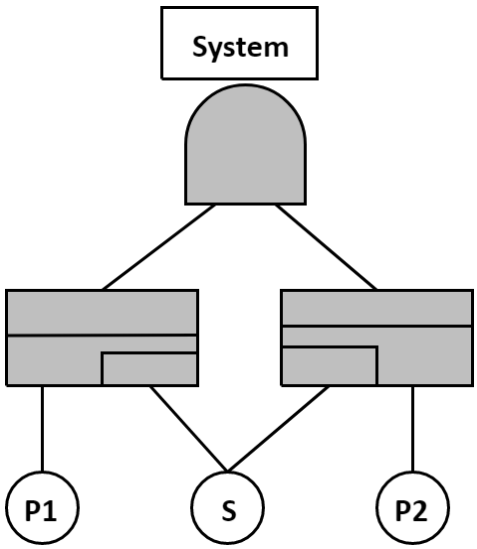


Figure 3.11: Spare gates sharing common spare resources

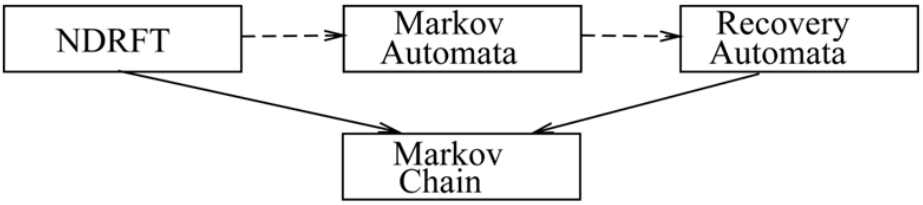


Figure 3.12: Transformation Flow

3.4 Synthesizing Recovery Automata from Markov Automata

The next step is to generate a scheduler for the Markov Automaton. The scheduler represents the best transitions that should be taken whenever the system faces non-deterministic actions. Based on the information from a scheduler, we can resolve the non-determinism in a Markov Automaton and generate the Markov chain. Markov chains are deterministic continuous-time Markov models. For the purpose of extracting the recovery strategy from a scheduler we need to formally represent this decision process and thus require Recovery Automaton. Recovery Automaton was introduced in 2.4.1.2 and we adapt it for the NDRFT. The thesis also adapts the algorithm used in NDDFT to generate the Recovery Automata for NDRFT. The Recovery Automata of a NDRFT is an automata that reads the occurred events (repair or fault) and outputs the next state and the recovery action sequence to be taken. The transitions in a recovery automaton can be either a Fault Transition or repair transition.

The input alphabet of the Recovery Automaton of NDRFT is the power set of repair and fault events $ES(T)$ and the output alphabet is the recovery action sequences $RS(T)$. Formally the Recovery Automaton for NDRFT is restated as:

Definition 7. (Recovery Automaton): A Recovery Automaton (RA) $R_T = (Q, \delta, q_0)$ of an NDRFT T is an automaton in which

- Q is a finite set of states,
- $q_0 \in Q$ is an initial state
- $\delta\colon Q \times ES(T) \to Q \times RS(T)$ is a deterministic transition function that maps the current state and an observed set of events to the successor state and a recovery action sequence.

The generated Recovery Automata have their transition labeled in the format:
$< MarkovianTransition >:< OptimalNon-DeterministicTransition >$.
Fig. 3.13 shows the general structure of a RA. Using the RA, the non-determinism can be handled in the NDRFT to generate the Markov Chain. The system can then be evaluated deterministically and also achieves better handling of the spare resources to improve system availability.

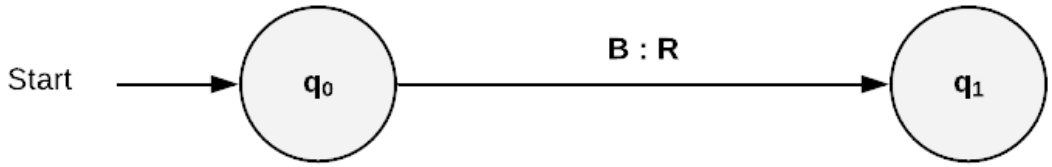


Figure 3.13: General Structure of a Recovery Automata

3.5 Minimization of Recovery Automata

Introduction of repair transitions and recovery actions lead to increase in state space of the generated automata. However, recovery automata usually have a lot of redundant states and transitions with empty transitions.By investigating the properties of the states of Recovery Automata we can reduce the state space size of Recovery Automata. In this section, we discuss the minimization algorithms that were adapted from NDDFT 2.4.1.2 in order to produce a more compact representation of the generated Recovery Automata. The minimization is done to remove redundant and non vital information from a Recovery Automata and produce equivalent Recovery Automata with fewer states and transitions.

The term recovery equivalence introduced in NDDFT captures the notion of two Recovery Automata having same behavior. We restate the definition of recovery equivalence between two Recovery Automata of NDRFT as follows:

Definition 8. (RA Recovery Equivalence). Let $R_1 = (Q_1, \delta_1, q_{01})$ and $R_2 = (Q_2, \delta_2, q_{02})$ be two RAs. We define a binary relation $\approx_R$ such that it holds true for any two RA that $R_1 \approx_R R_2$ iff for any sequence of sets of events $E_1, \ldots, E_n$ it holds that:

$$Recovery_{R_1}(E_1, \ldots, E_n) = Recovery_{R_2}(E_1, \ldots, E_n)$$

given that the for all $i \in N$, $E_i \in ES(T)$ where $N = \{0, 1, 2, \ldots$ and $ES(T) = BES(T) \cup RES(T)$.

3.5.1 Trace Equivalence

The previous work in NDDFT also employed the trace equivalence for the minimization of Recovery Automata. The algorithm can also be used for the Recovery Automata generated from NDRFT as well. The definition of Trace Equivalence had been lifted to the states of Recovery Automaton in [MMGN18]. We redefine this definition to cover the Recovery Automata of NDRFT as follows:

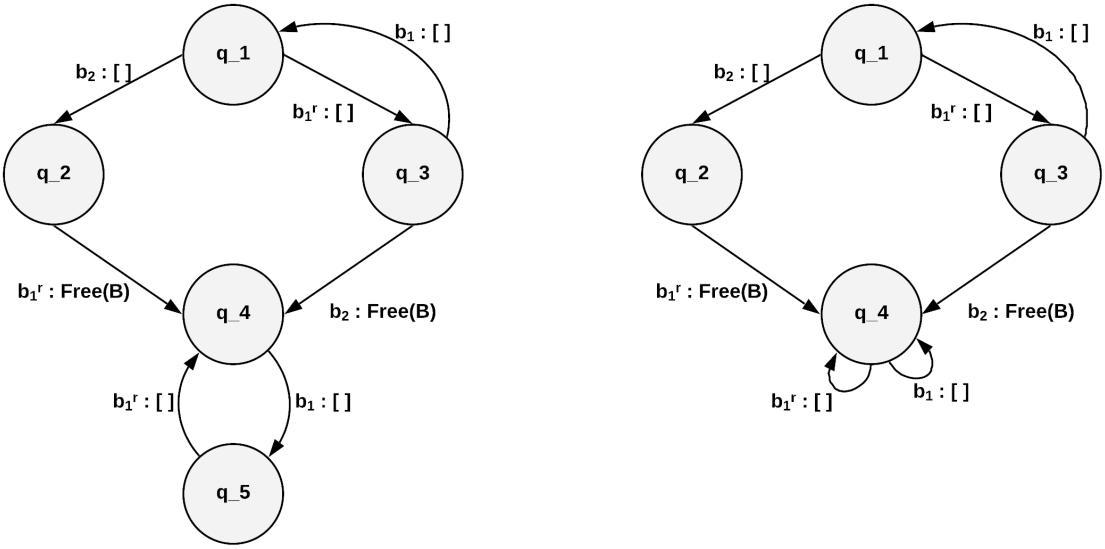


Figure 3.14: A Recovery Automaton a) before and b) after applying the Partition Refinement with Trace Equivalence

Definition 9. (Trace Equivalence). Let $R_T = (Q, \delta, q_0)$ be an RA. A trace equivalence $\approx \subseteq Q \times Q$ is a maximal binary relation such that it holds for any states $q_1, q_2 \in Q$ that $q_1 \approx q_2$ iff for any $E \in ES(T)$ it holds that:

$$\delta(q_1, sE) = (q_1', rs_1) \text{ and } \delta(q_2, E) = (q_2', rs_2) \text{ with } q_1' \approx q_2' \text{ and } rs_1 = rs_2$$

Partition Refinement Algorithm with the trace equivalence concept is used to identify equivalent states. The equivalent states are then merged to produce smaller Recovery Automata with equivalent recovery behavior. Fig. 3.14 shows an example output of the application of this rule with NDRFT.

Additionally, the previous work also included algorithm to combine non-trace equivalent states to yield a smaller Recovery Automaton introduced in section 2.4.1.2. It works on the concept of removing untakeable transitions and combining states that are orthogonal with respect to its guaranteed inputs. Untakeable transitions are those traces in a Recovery Automata that can never occur and are thus considered invalid. Both the methods of removing untakeable transitions and merging orthogonal states work on the assumption that all events are permanent and that an event can occur only once. However, this condition does not hold true with repairable fault trees as some events can occur more than once in a repairable system.

In the adapted algorithm for Orthogonal minimization of Recovery Automata generated from NDRFT, we calculate the active faults for each state instead of the guaranteed inputs. Guaranteed inputs calculated the guaranteed components to have failed in a given state based on the inputs to the state. In a NDDFT, the guaranteed inputs reflected the actual components

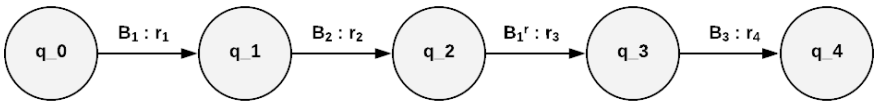


Figure 3.15: Section of Recovery Automata

failed since only failure events can occur and all events are permanent. However, with the NDRFT calculating guaranteed inputs fail to reflect the components failed in a state, since not all events are failure events or permanent events. It fails to capture the current failure status of the system since repair transitions can change the fail status of the components. For this reason, we need to calculate the active faults in an NDRFT by identifying repair transitions and updating the set of failed components. In the Fig. 3.15, we can see a sequence of a Recovery Automata with failure and repair transitions. The guaranteed inputs for the state q_4 is B_1, B_2, B_1^r, B_3 and the active fault of the state is B_2, B_3.

In the following subsections we discuss the calculation of the active faults and the adapted orthogonal merging rule that will be used in the thesis work.

3.5.2 Active Fault

Active Faults represent the failure status of the components in a system at a given state. Computation of active faults is calculated from the incoming transitions to a state. Failure transitions add the component to the list of failed components and repair transitions remove it from the failed set. The initial state of a Recovery Automaton always has no active faults since all the components are considered operational and without faults in the initial state. The active faults are calculated by employing the work-list algorithm [Kil73] using the transfer function given below:

$$AF(q_0) := \emptyset$$

$$AF(q) := \begin{cases} \bigcap\limits_{(p,E)\in pred(q)} AF(p) \cup \{E\}, & \text{when } E \in BES(T) \\ \bigcap\limits_{(p,E)\in pred(q)} AF(p) \setminus \{F(E)\}, & \text{when } E \in RES(T) \end{cases}$$

where $pred(q) := \{(p,E) \mid \delta(p,E) = (q,rs) \text{for some } rs, p \neq q\}$ denoting the set of predecessor transitions of a state q and F(E) is the mapping of a repair event to its equivalent Fault event in the set $BES(T)$. The calculated active faults are next used in removing untakeable transitions and in the orthogonal rule in the next steps.

3.5.3 Removing Untakeable Transition

The previous work of removing untakeable transitions is based on the property that a basic event can occur only once and if we can confirm that the basic event has occurred in every path leading to the state, it can never occur again in the future. In the adapted version, a transition is considered invalid and can be removed without affecting the Recovery Automaton, based on the active faults in a particular state. We calculate the set of active faults for a given state and then identify transitions that can not occur for the given state. A state having an active fault of a particular component can discard any failure transitions of the component in that state. Similarly, when there is no active fault of the component for a given state, no repair transition is expected to take place and we can safely remove repair transition of the associated fault. In this way, the invalid traces are identified and safely removed without affecting the recovery behavior of the automata.

In the Fig. 3.16, we can see that the untakeable transitions in state q_1 and q_2 are $B_1 : \varepsilon$ and $B_2 : \varepsilon$ since they have an active fault of B_1 and B_2 in their respective states and are thus removed in the next step as depicted in the figure.

The next step after eliminating invalid traces from a Recovery Automata is identifying orthogonal states. The orthogonal states are identified using the criteria given below. After the identification, the orthogonal states are merged. The concept of orthogonality in an NDRFT extends the original orthogonal rule to merge non-trace equivalent states. Formally, the orthogonal states in the Recovery Automata of a NDRFT can be defined as below

3.5.4 Merging Orthogonal States

Definition 10. (Orthogonal States). Let $R_T = (Q, \delta, q_0)$ be an RA. Let $p, q \in Q$ be two non-initial distinct states and $B \in BES(T)$. Then p, q are orthogonal with respect to B iff

$$B \in AF(p) \cup AF(q)$$

The definition of orthogonality is illustrated with the example as shown in Fig. 3.16. The Recovery Automata reacts to event sets $\{B_1, B_2, B_2^r\}$ where $\{B_2^r\}$ is the repair event set for $\{B_2\}$, with associated recovery actions $\{rs_1, rs_2, rs_3\}$. The different event sets involved are

- $BES(T) = \{B_1, B_2\}$,
- $RES(T) = \{B_2^r\}$ and
- $ES(T) = BES(T) \cup RES(T) = \{B_1, B_2, B_2^r\}$.

Based on the sets of events and the active fault definition 3.5.2, the active faults for each state is calculated as follows:

- $AF(q_0) = \emptyset$,
- $AF(q_1) = AF(q_0) \cup \{B_2\} = \{B_2\}$,
- $AF(q_2) = AF(q_0) \cup \{B_1\} = \{B_1\}$ and
- $AF(q_2) = (AF(q_1) \cup \{B_1\}) \cap (AF(q_2) \cup \{B_2\}) = \{B_1, B_2\}$.

Based on the calculated active fault and orthogonal rule, it holds that states q_1 and q_2 are orthogonal with respect to active faults B_1 and B_2. The orthogonal states are merged to produce the equivalent state q_12 and all the incoming and outgoing transitions of the original states are redirected to the merged state as shown in the figure.

The Orthogonality concept can now be extended with the trace equivalence definition to redefine the Recovery State Equivalence from the original work as follows:

Definition 11. (RA State Recovery Equivalence). Let $R_(T) = (Q, \delta, q_0)$ be an RA. A state-based recovery equivalence $\approx_R \subseteq Q \times Q$ is a maximal relation such that it holds for any state $q_1, q_2 \in Q$ that $q_1 \approx_R q_2$ iff for any $E \in ES((T))$ it holds that either:

- $\delta(q_1, E) = (q_1', rs_1)$ and $\delta(q_2, E) = (q_2', rs_2)$ with $q_1' \approx q_2'$ and $rs_1 = rs_2$ or
- q_1, q_2 are orthogonal with respect to B.

3.5.5 Merging Final States

The algorithm also uses the rule of merging fail states discussed in 2.4.1.2. The rule states that if there is fail state that does not contribute to any new recovery actions, the transition

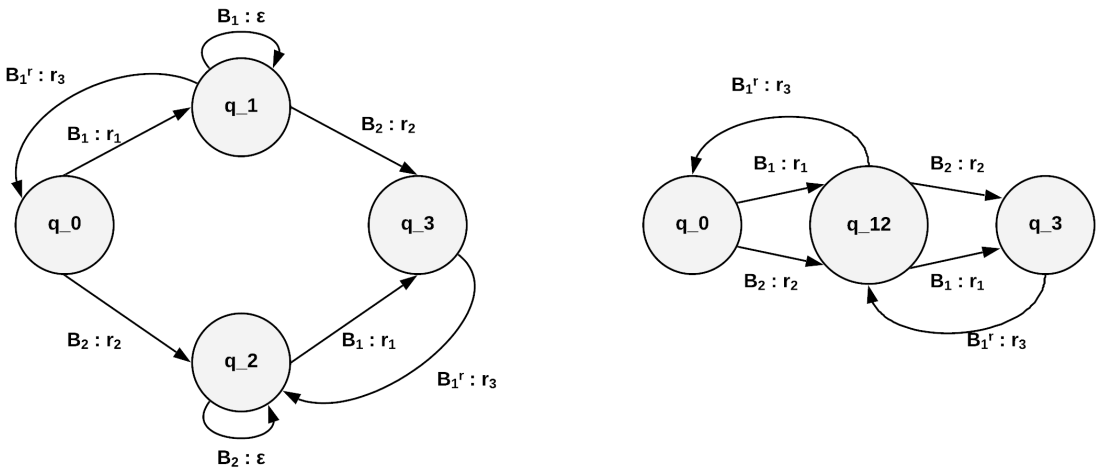
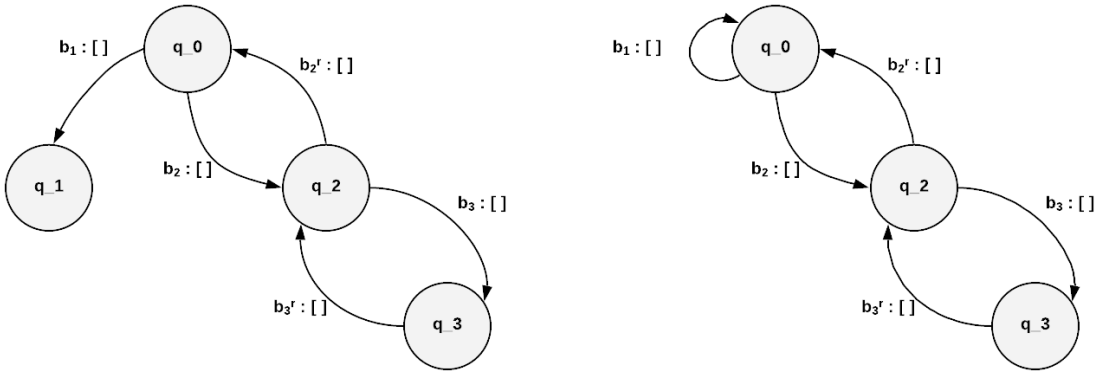


Figure 3.16: A Recovery Automaton a) before and b) after applying the adapted orthogonal state rule

from its previous state can be made into a self loop. The state can then be eliminated if it is unreachable.

However, this rule considers all fail states to be absorbing which is true for NDDFT, but not for NDRFT. The rule of merging the fail state to its predecessor state is not applicable when the fail states have outgoing repair transitions that lead them to other states. Since some fail states may carry valid recovery sequence information, the rule is not applicable for fail states that are not absorbing states. This restricts the minimization effect of the rule for merging fail states. The rule is still used in the minimization for the Recovery Automaton for NDRFT to remove fail states that conform to the definition of FAIL states in 4.

In the Fig. 2.14, we can see a Recovery Automaton with both absorbing fail state and non-absorbing state. Applying the rule, we can see that the Recovery Automaton can be reduced to some extent when the system has both repairable and non-repairable components.



(a) Before Rule

(b) Applying the rule

Figure 3.17: A Recovery Automaton a) before and b) after applying the Final state rule

Chapter 4

Implementation

This Chapter describes the implementation of the proposed concept. The following sections briefly describe the development framework and an overview of the implementation of the discussed concepts. The section end with a brief description of the implementation of the unit testing done.

4.1 Development Environment

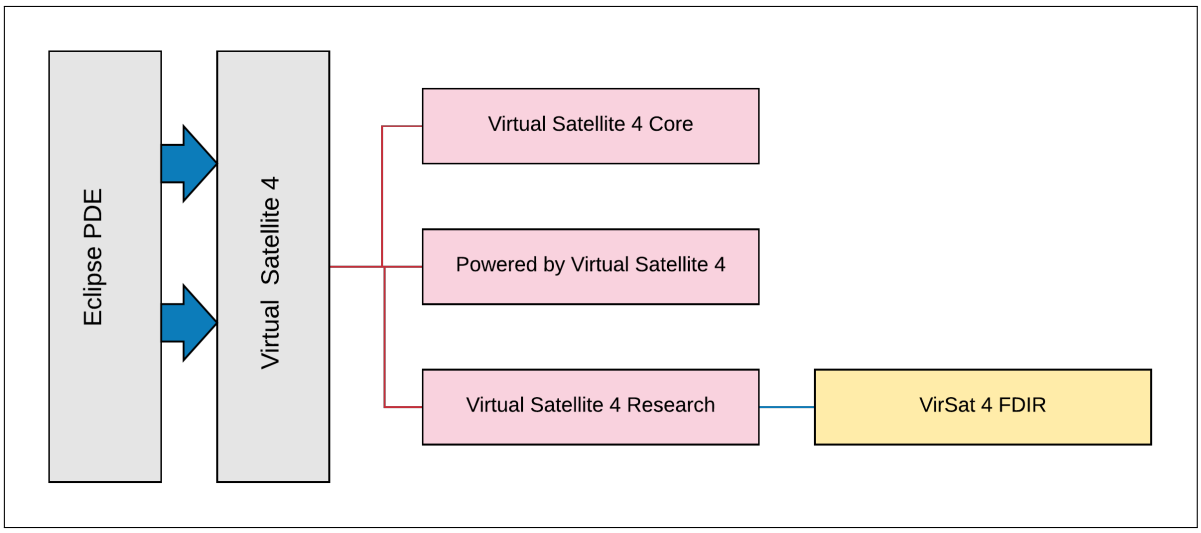


Figure 4.1: Virtual Satellite 4 Framework

A proof of concept is implemented in Java 8 in the Eclipse plug-in Development Environment of the Virtual Satellite FDIR application to provide a proof of concept. Virtual Satellite 4 FDIR is based on the Virtual Satellite 4 framework (VirSat). VirSat is a framework developed at German Aerospace Center (DLR) to support Model Based System Engineering for the entire life cycle of a satellite mission. It is developed in the Java programming language built on the Eclipse Rich Client Platform (RCP). The Eclipse RCP supports the develop-

ment of plug-in-based applications. The Virtual Satellite 4's plug-in based framework and its flexible data model can be extended to implement specific application to support different projects and engineering processes. The extensions are done through *Concepts* that provide the corresponding user interface and basic functionalities for further development. The Concepts are delivered as an Equinox Feature containing plug-ins and it describes the extensions to the core data model. Based on the Concept extension, there are three product lines:

1. **Virtual Satellite 4 Core**: It is the core application of the VirSat Framework and provides basic functionality needed to design a space system. Through the concepts and Virtual Satellite development tools, advanced functionalities can be added.

2. **Powered by Virtual Satellite 4**: These are the specific applications that were extended from Virtual Satellite for different Projects. The functionalities are tailored to meet the requirements of the requirements of specific projects.

3. **Virtual Satellite 4 Research**: These applications are based on the Virtual Satellite Core created for research and experimentation for individual applications. Since they are separated from production line, this provides an interface to perform experimentation for scientific research. One of such application is VirSat FDIR.

   VirSat 4 FDIR is an extension with the Fault Detection, Isolation and Recovery concept. The features supported by VirSat 4 FDIR include Fault Modeling of a system, Quantitative analysis of the fault trees, modeling recovery strategies and synthesizing recovery strategies. The VirSat FDIR has implementation for Non-Deterministic Dynamic Fault Trees, introduced in section 2.4.1 in addition to the Standard Dynamic Fault Tree implementation. In the thesis work, the implementation of the proposed concepts of the Non-Deterministic Repairable Fault Tree is done in the VirSat FDIR 4 framework to support NDRFT semantics in addition to the existing fault tree semantics. The Fig. 4.2 shows the graphical interface of the VirSat 4 FDIR application.

4.2 Architecture

The Fig. 4.3 shows the overall updated component diagram of the Fault Tree Analysis section of the VirSat FDIR. It shows the sections involved in the system analysis of the fault tree. The fault Tree component represents the input to the system which is either given through the graphical tool or read from the Galileo DFT file [SDC99]. The DFT to MA converter takes as input the semantics and fault tree and then perform transformation of the fault tree to Markov Automata. The Recovery Automata synthesizer component is dependent on the DFT to MA converter to get the Markov Automata and produce the Recovery Automata. The Fault Tree

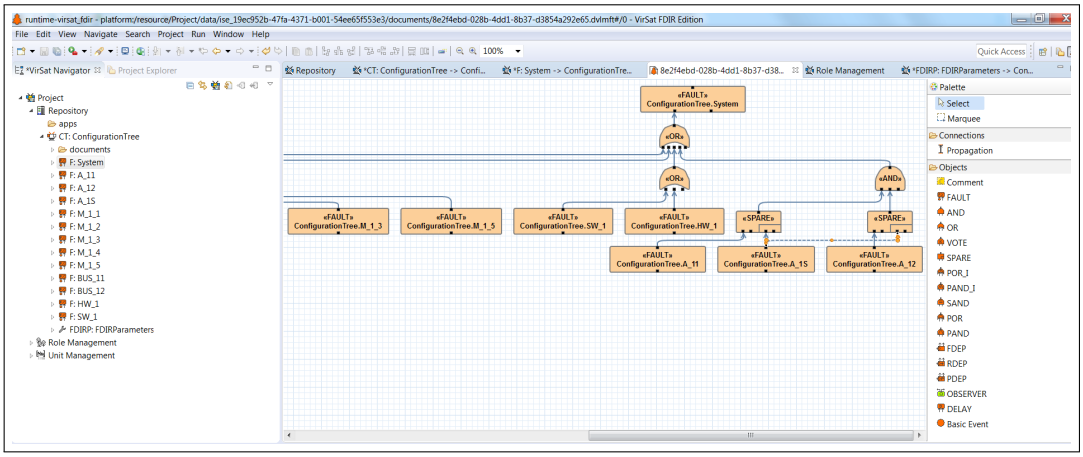


Figure 4.2: VirSat FDIR Tool

evaluator requires the Markov Automata and Recovery Automata to generate Markov chain and perform fault tree analysis.

The thesis work is responsible for introducing the component NDRFT semantics to the architecture. The DFT to MA converter can now read the NDRFT semantics in addition to previous semantics. The Thesis work also modifies the already existing components: DFT to MA and Recovery Automata to enable the support for the NDRFT.

Fig. 4.4 represents the activity diagram of the system analysis process using the NDRFT semantics.

4.3 Gate Semantics

Below, the changes done in the implementation of the Gate semantics to support Repairable systems with non-deterministic recovery actions is discussed. Fig. 4.5 represents the type hierarchy of the different gate semantics. The FDEP Semantics and NDRSpare Semantics were implemented for the thesis work. The other gate types involved adapting the already existing implementation to include both repair and fail transitions.

4.3.1 Static and Priority Gates

The implementations for Static and Priority gates were reused from the NDDFT implementations. Based on the type of transition: Repair or Failure, the fail event list is updated through the Markov Automata generation method.

4.3.2 FDEP Semantics

The FDEP Semantics needed to reimplemented for the case of NDRFT since now we need to track the actual dependent events triggered by FDEP to perform repair on only affected com-

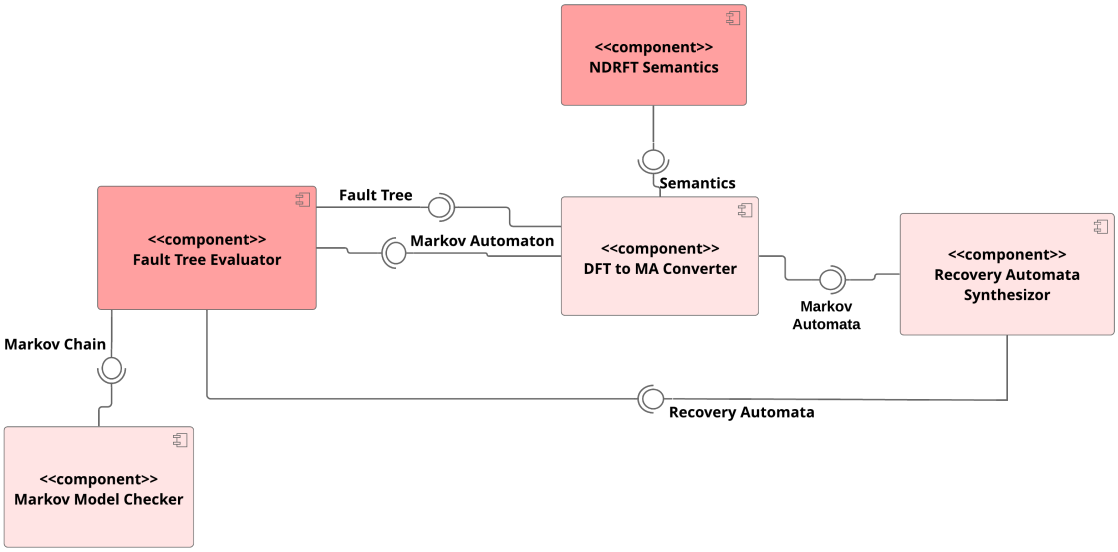


Figure 4.3: Component Diagram of the System

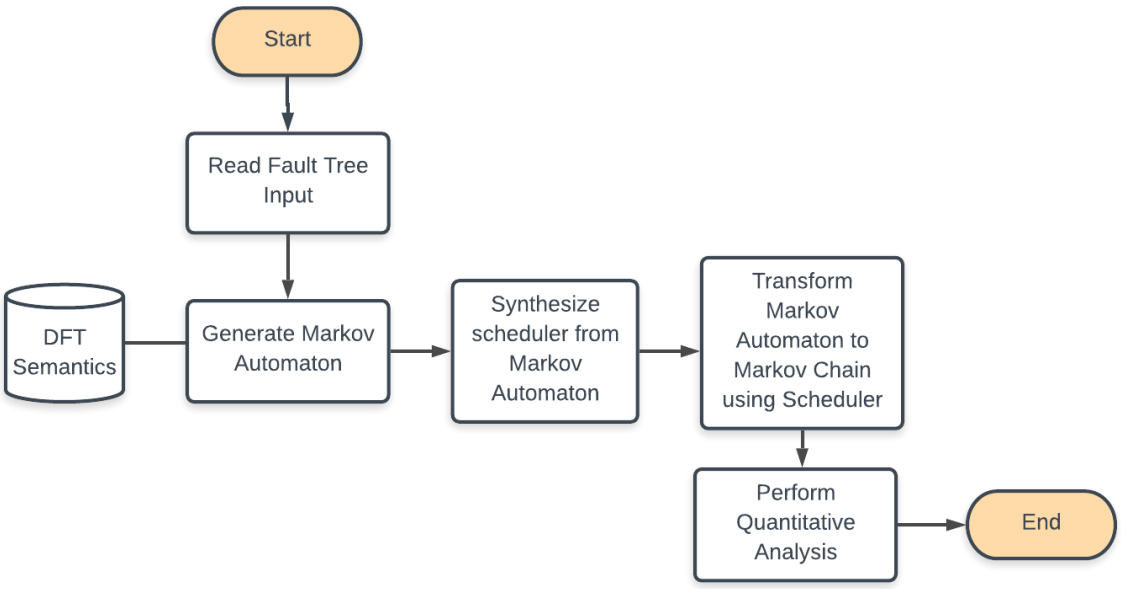


Figure 4.4: The activity diagram of Fault Tree Analysis with NDRFT

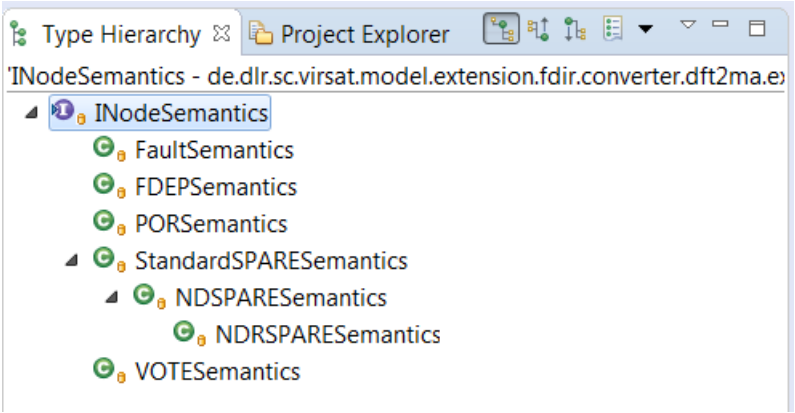


Figure 4.5: Semantics Heirarchy in Implementation

ponents. In the previous case, the implementation for handling FDEP update was handled in Fault Semantics as it only involved transferring the failure from trigger to dependent events and it did not matter in the NDDFT semantics whether the dependent events failed due to the trigger event or some other cause.

In NDRFT, FDEP gates propagate the failure as well as repair to its dependent events on the occurrence of its trigger event. A Trigger event to a FDEP may be either the failure or repair of a component. To implement the proposed semantic, a map to store dependent events that failed due to the FDEP trigger was used. This is needed to differentiate between failed components that had already failed due to other fault events and failed components that failed due to the FDEP trigger. This distinction is needed in the repair transition step where the dependent events that were triggered by the FDEP are restored. Also, when the FDEP Trigger is in fail state, the repair of its dependent events cannot take place regardless of whether the failure of the dependent event was caused by the FDEP trigger or a previous fault.

The pseudocode describing the process for updating the fail status with FDEP gate is

given below:

Algorithm 1: FDEP Event Update

```
1  T:= get child of FDEP;
2  DE:= get dependent events of FDEP;
3  FS:= get Current failed event;
4  FT:= get FDEP triggered set;
5  if T has Failed then
6      Add T to FS;
7      for ∀E ∈ DE do
8          if E is not in FS then
9              Add E to FS; Add E to FT;
10         end
11     end
12 else
13     Remove T from FS;
14     for ∀E ∈ FT do
15         if E is in FS then
16             Remove E from FS; Remove E from FT;
17         end
18     end
19 end
```

4.3.3 SPARE Semantics

The implementation for the proposed spare gate semantics involved extending the NDSPARE Semantics to override the method to update the process to be followed in case of a failure status update. It also includes the implementation for Free action. The free action removes the claimed spare from the map of claimed spare resources and deactivates the spare resource. The generated state from free action is a non failed state and represents the state where the

spare gate uses the primary input.

Algorithm 2: Spare Event Update

```
1  P:= get primary of SPARE;
2  S:= get secondary events of SPARE;
3  FS:= get current failed event; if P or S has failed then
4  |   Add P or S to FS;
5  |   for ∀E ∈ S do
6  |   |   E is not in FS Perform CLAIM(Spare,S);
7  |   end
8  else
9  end
10 P is Repaired CS:= Get Currently Claimed Spare;
11 Perform Free(CS);
12 else if S is Repaired then
13 |   Currently claimed Spare is failed Perform Claim(S);
14 Perform Empty();
```

4.4 Algorithms

The implementation for NDRFT required adapting the algorithms implemented for NDDFT for generating the Markov Automata and Recovery Automata. The below sections describe the changes made in the algorithm implementation to support NDRFT semantics.

4.4.1 Markov Automaton Generation Algorithm

The Markov Automaton generation algorithm that was used for Non-Deterministic Dynamic Fault Trees was reused. The Markov Automaton generation algorithm includes the following steps:

1. The algorithm starts with the initial state and generates the next states and successor transitions based on the calculated occurable events for the state and the DFT semantics.

2. A repair event for a component is included in the set of occurable events when the component has already failed and has a repair rate assigned to it. A fault event is included in the set of occur able events for a state if the related component is functioning at the given state.

The algorithm also checks for the type of gate that is affected by the event and generates the next state based on the updated gate semantics for repairable fault trees. It also updates the changes in the parent nodes of the affected gates propagating upwards to the top level events in case the child causes a change in state of its parent. Each state also carries information regarding the active faults in the given state which is a representation of currently failed components and an additional information of nodes affected by FDEP trigger and spare resources claimed by the SPARE gate as required by its semantics. In every state the list of occurable events is based on the active faults of the state. A repair event removes the associated active fault from memory and a failure event adds the component to the list of failed components. tarting with the initial state, the algorithm executes the list of occurable events to produce the next state based on the DFT semantics. The same step is iteratively repeated for the generated states until no new states can be produced. The generated states are also checked to see if an equivalent state already exists and is merged with the previously existing state.

4.4.2 Recovery Automaton Synthesis Algorithm

Recovery Automaton is synthesized from the Markov Automaton by using the adapted value iteration algorithm from [GHH+13] which calculates the expected time to reach the goal states. Computation of maximum expected time reachability in the Markov Automaton is done using the below formula [GHH+13] where MS is the Markovian state, PS is the probabilistic state, S is the set of states, $v(s')$ is the reward to reach s', $Act(s)$ is the set of actions from s, $\mu(s')$ is the distribution over the action from $s \to s'$, $P(s,s')$ is the probability to move from $s \to s'$ and $E(s)$ is the exit rate of the state. :

$$[L(v)](s) = \begin{cases} \frac{1}{E(s)} + \sum\limits_{s' \in S} P(s,s').v(s'), & \text{if } s \in MS \\ \min\limits_{\alpha \in Act(s)} \sum\limits_{s' \in S} \mu(s').v(s'), & \text{if} s \in PS \end{cases}$$

The transitions in a Markov Automata are either probabilistic transitions or Markovian transitions and based on the outgoing transitions states may be either probabilistic states PS or Markovian state MS. Unlike the algorithm in NDDFT, the failure states in NDRFT are not absorbing and thus the expected time for the outgoing transitions from the failure state was calculated as well. For a Markovian state s, the expected time to reach the failure state is the sojourn time in state s plus the expected time to reach the failure states through its successor states. For probabilistic states with multiple actions, the transition with the maximum time required to reach the failure state was chosen. Thus, transitions that allow us to stay in non-fail state are assigned higher rewards. The states are assigned scores based on the outgoing transitions. As a final step, the Recovery Automaton is generated by choosing transitions that lead to a better valued state.

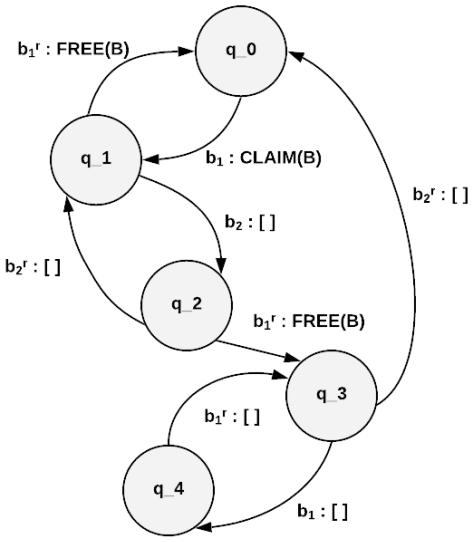


Figure 4.6: A Recovery Automaton of a 2 input SPARE

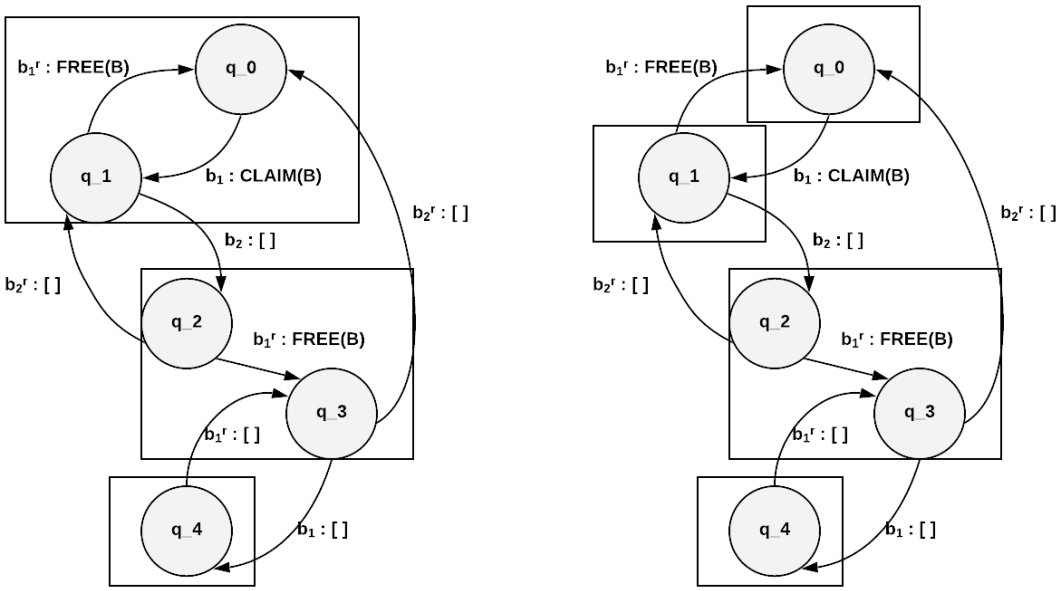
The figure 4.6 shows a Recovery Automaton that was generated from a spare gate with two repairable inputs A and B. The transitions are labeled as $< transition >:< RecoveryAction >$. The transitions can be labeled as either F($< component >$) or R($< component >$). The states in a Recovery Automaton are named as $q_0, q_1 \ldots q_n$.

4.4.3 Minimization Algorithm

The implementation of the minimization algorithm using the concept of Orthogonal states was modified to reduce the states for Recovery Automata generated from repairable systems. Based on the proposed concept the minimization algorithm requires calculation of active faults for each state since events associated with repairable components can occur more than once. The following section describes the implementations for calculating active faults and merging states that are orthogonal. The Orthogonal Refinement Minimization algorithm is adapted to support repair by calculating active faults instead of guaranteed inputs. The next state of using partition refinement is also modified to support using the active faults for determining equivalent states and handle repair transitions.

4.4.3.1 Computation of Active Faults

The active faults in each state are calculated from the incoming inputs to the state. The transitions that are backward transitions and non-backward transitions were differentiated. Backward transitions are those that come from a successor state to one of the previous states. The backward transitions do not contribute to the current active fault of a given state. Based on this, the active faults for an initial state is always empty despite having incoming transi-



(a) Initial blocks

(b) Refined blocks

Figure 4.7: Orthogonal refinement algorithm on Recovery Automaton of a 2 input SPARE

tions as they are all backward transitions and are always repair transitions.

The concept from 3.5.2 for calculating active fault for the given state was used. The incoming transitions are checked if they are repair. Only the Fault event transitions are added to the set of active faults. When the transition is repair, the corresponding active fault is removed. The results are then stored as a map mapping of state s $\rightarrow$ Active Fault AF.

4.4.3.2 Merging Orthogonal States

The implementation for merging orthogonal states is adapted to support repair. The previous implementation of using the Partition refinement algorithm with concept of orthogonality to find orthogonal states was employed. The figure shows the process of grouping states into blocks that are potentially equivalent based on active faults and outgoing transitions and then refining the blocks to remove non-equivalent states. In the Fig. 4.7 the Recovery Automaton of a 2-input spare with repairable components are considered. The initial blocks are $[q_0, q_1]$,$[q_2, q_3]$ and $[q_4]$. After the refining step, the blocks are $[q_0]$, $[q_1]$, $[q_2, q_3]$ and $[q_4]$. The states in the blocks are merged in the next step where the incoming and outgoing transitions of the states of the block are directed to and from the new merged state.

The algorithms shown takes as input the Recovery Automaton RA, the calculated active fault AF, Mapping of state to Outgoing transition O, Guard Profile GP which stores a mapping of the state to outgoing transitions with recovery transitions.

Algorithm 3: Orthogonal Minimization

Data: RA, AF, O, GP

Result: Reduced Recovery Automaton

1 : $P := A_1, \ldots, A_n$ such that state s $\in$ A is Orthogonal w.r.t AF;
2 **while** $\exists\, A \in P$ *such that A is splittable* **do**
3 split A into $A_1, \ldots, A_n$;
4 $P := P - A \cup \{A_1, \ldots, A_n\}$;
5 **for** $\forall A \in P$ **do**
6 Merge states in A;

4.5 Implementation of Unit Test cases

The implementation of the semantics was tested using unit tests. They test small functionalities or part of the program for expected behavior. The below sections describe the input format and the unit testing involved in the work.

4.5.1 Input Format

The input fault tree model was given in the Galileo format [SDC99]. It allows representation of the dynamic fault trees in a textual format. A Galileo specification always begins with specifying the top level failure as: **toplevel** $< name >$;. It is then followed by description for each node of the fault tree. The general format for representation of gate is:
$< name >< Gate >< input_1 >< input_2 > \ldots < input_n >$;.

The basic events are represented as:
$< name >< attribute_1 >=< value >< attribute_2 >=< value > \ldots < attribute_n >=< value >$
The attributes can be lambda that represents failure rate, repair to indicate repair rate and dorm that indicates the dormancy factor of the spare resources. The files are stored as .dft files and are read during run time for fault tree analysis. An example DFT Galileo format of the pump system is given below:

```
toplevel "System";
"System" and "OR1" "OR2";
"OR1" or "PAND1" "SPARE1";
"PAND1" pand "B1" "B2";
"SPARE1" csp "B2" "B3";
"OR2" or "PAND2" "SPARE2";
"PAND2" pand "B4" "B5";
"SPARE2" csp "B5" "B3";
```

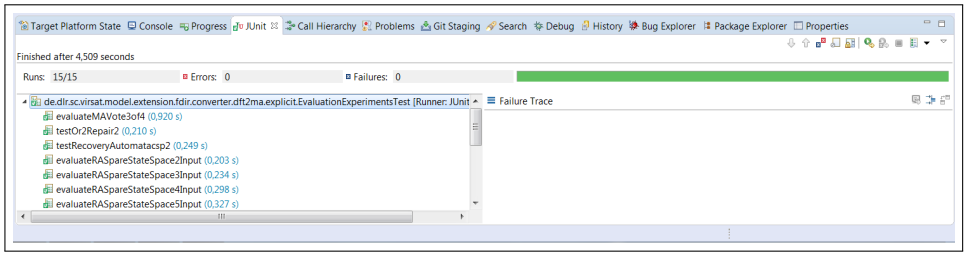


Figure 4.8: JUnit Test run

```
"B1" lambda=3 repair=1;
"B2" lambda=6 repair=2;
"B3" lambda=4 repair=6;
"B4" lambda=2 repair=5;
"B5" lambda=6 repair=2;
```

4.5.2 JUnit Tests

JUnit Tests were created for validating the implementations. The unit tests are used to validate the code for expected results. The unit tests are realized using the JUnit Tests that provide assert methods to compare expected results with the actual results. The algorithms for Markov Automaton, Recovery Automaton synthesis and Minimization were tested by checking the state space size for the generated automaton for different gates. A visual analysis of the generated automaton for smaller inputs is also done as we know the correct output for each gate. Furthermore, the calculated availability values were also compared with the deterministic approach to show that this approach produces an improvement in the availability of the system. The test cases were written to check behaviour of each gate as well as outputs of commonly used gate combinations. The Fig. 4.8 shows the test cases and test runs. As can be seen from the figure, the test runs have been successful.

The percentage increase in the Recovery Automaton and Markov automaton state space size is also compared with the introduction of repair. The table below shows the comparison for each gates for the case of 2 inputs with a NDDFT and NDRFT semantic. The state space size for MA and RA is the same for all gates except the SPARE due to non-determinism in SPARE. The SPARE shows an increase of almost 3 times the state space size of a NDDFT. Thus, introduction of repair shows a significant amount of state space growth for both Markov Automata and Recovery Automata.

Gate	NDDFT	NDRFT	% Increase
AND	4	4	0
OR	2	4	100
PAND/POR	5	5	0
FDEP	4	2	100
SPARE	5(MA) 3 (RA)	14(MA) 5(RA)	180(MA)) 67(RA)

Table 4.1: Comparison of number of states generated for individual 2 input gates with repair

Chapter 5

Evaluation

This chapter describes the various aspects relating to how the implementation was evaluated. The following sections describe the system specification where the evaluations were done, the input format of the fault tree data and case study examples to evaluate the results of the implementations and a discussion of the results.

5.1 System Specification

The experiments are carried out on a system with the following specifications:

- **Processor**: Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz 2.70 GHz
- **Installed Memory (RAM)**: 16 GB
- **System type**: 64-bit Operating System

5.2 System Evaluation

The implemented semantics and algorithms were evaluated to check the following aspects:

- Correctness of semantics
- Effectiveness of minimization rule
- Measurement of system metrics
- Improvement in availability of the system using the Non-Deterministic approach in comparison to the Dynamic Repairable Fault Tree.
- State space growth and reduction.

The following sections discuss in detail the experiments done to test the given points.

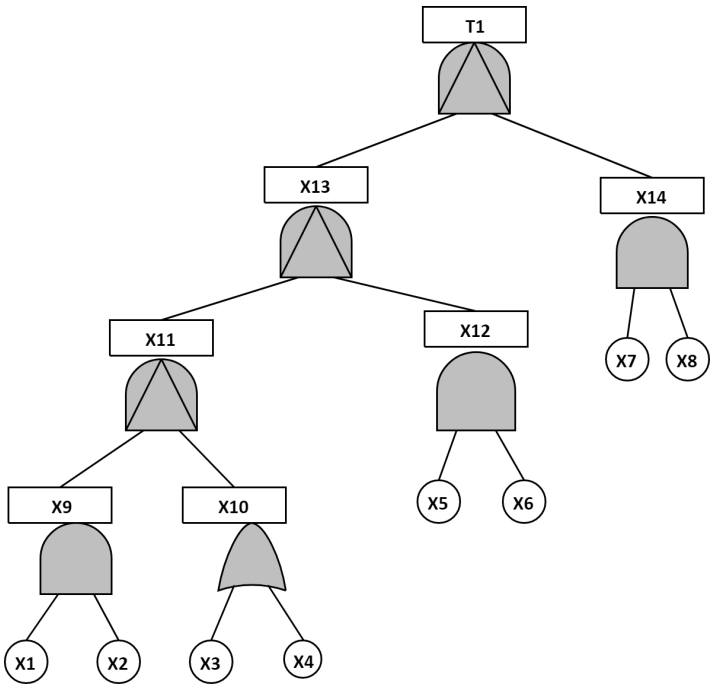


Figure 5.1: Sub-System of Binary Hypercube architecture

5.2.1 Test Verification of Semantics

The semantics was verified for its correctness by comparing with the results of Temporal Dynamic Fault Trees [AHMS19]. As introduced in previous section 2.3.4 [AHMS19], the version of fault tree considers transient faults and has semantics for all gates without the spare gate. The case study from the paper that uses an adapted extended fault tree of a external event to the Binary Hypercube Architecture is used for this experiment.

5.2.1.1 Case Study: Binary Hypercube Architecture

The Binary Hypercube architecture example was originally taken from [DBB92]. It is a common computer architecture for parallel computing using a network of processors. The Fig. 5.1 shows the fault tree of a component of binary hypercube architecture used for the analysis. The example uses the PAND, AND and OR gates and thus only the comparison for the given gates can be performed. The case study considers different sub-systems X9, X12 and X14 repairable separately and measures the effect of it in on availability. The basic events have been assigned fail rates of 0.005 and the repair rates are assigned 0.33

5.2.1.2 Results

The results shown in the table 5.1 compares the values of the availability calculated for the implementation with TDFT with the calculated availability of NDRFT for different repairable sub-systems. The results how that the availability values are close and thus, it can be con-

	NDRFT	TDFT
X9	99.999	99.997
X12	99.998	99.893
X14	99.83	99.8

Table 5.1: Test Verifying Semantics

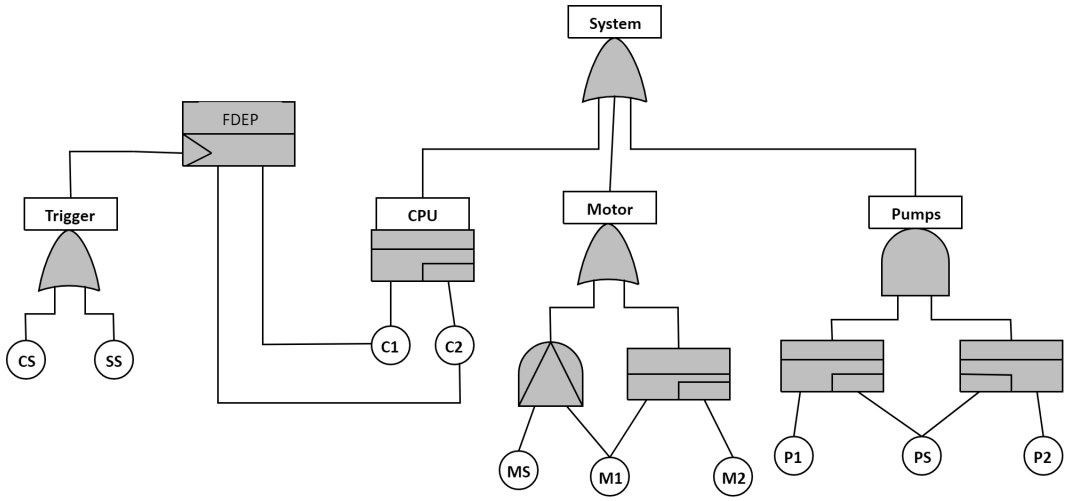


Figure 5.2: Cardiac Assistant System

cluded that the semantics implementation of the NDRFT is correct.

5.2.2 Test Verification of Minimization

The correctness of the minimization algorithms can be tested by evaluating the system with a unminimized recovery automata and a minimized recovery automata and then comparing the results of availability. It can be concluded that the minimized and unminimized recovery automata have the same recovery information when both the automata can produce the same results. For the purpose of this experiment, the use case study of Hypothetical Cardiac Assistant System is considered.

5.2.2.1 Case Study: Hypothetical Cardiac Assistant System

The example of a Hypothetical Cardiac Assistant System was used for the evaluation of the dynamic fault tree. The system comprises of four subsystems: CPU unit, Motors, pumps and Trigger Unit. The system can fail if any of the four subsystems fail, thus using a OR to represent this relation. The CPU Unit has a primary CPU P and backup CPU B which gets activated when the primary fails. It is thus represented using a Spare gate. The Motor unit required at least one of the motors to work to require functioning of the system. The Pump system uses redundant pumps Pump 1 and Pump 2 and they both share a common backup

-	States	Transition
Unminimized	919	2280
Minimized	69	449

Table 5.2: Minimization

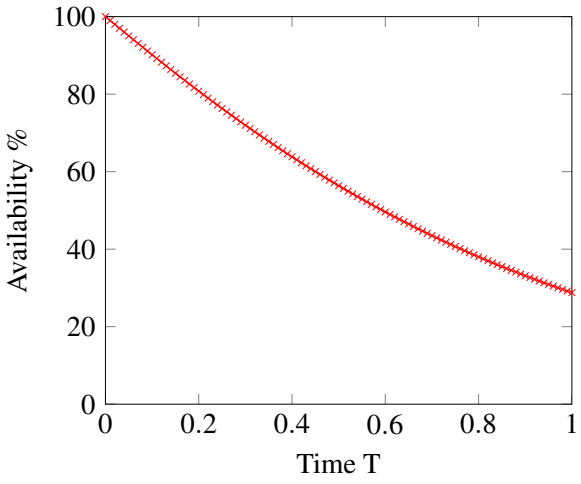


Figure 5.3: Availability

pump which will replace them when they fail. A pump unit fails when all the three pumps fail. The crossbar switch (CS) and system supervisor (SS) can trigger the CPUs to shutdown. This is represented using the FDEP gates showing the functional dependencies of the CPU unit with the trigger system. The figure shows the dynamic fault tree construction of the HCAS.

In this experiment, we consider the CPU unit as repairable and compare the calculated availability before and after minimization.

5.2.2.2 Results

The table 5.2 shows the comparison of the state space size before and after minimization and the availability of the system. There was a reduction of 92.4% for the states and 80.3% for the transitions. The system was evaluated to calculated the availability upto T=1. Since, both the minimized recovery automata and unminimized recovery automata showed the same results, we can conclude that the minimization can reduce the state space size without altering the recovery automata behavior.

5.2.3 Measurement of system metrics

For verifying the improvement in system metrics, the same case study example of HCAS from 5.2.2.1 was used. The implementation was executed and system analysed to compare the values of availability and mean time to failure. The experiments are run with the CPU unit as repairable and also without consideration for its repair. It is expected that the availability of the system improves with the introduction of repair.

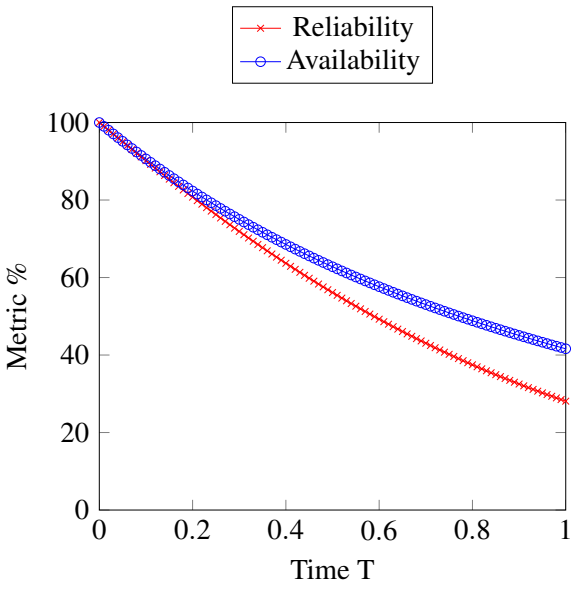


Figure 5.5: Measured System Metrics of HCAS

5.2.3.1 Case Study: Hypothetical Cardiac Assistant System

The description of the system is discussed in previous section 5.2.2.1. The basic events are assigned fail rates of 0.5 and the repair rate 1 is assigned additionally to the CPU unit.

5.2.3.2 Results

Fig. 5.5 shows the calculated reliability and availability for the system upto time point T=1. The mean time to failure was 0.706 and the system reaches a steady state availability of 0 since the other sub-systems of the system ae not repairable and when any of the basic event occurs the system reaches a permanent failed state.

5.2.4 Comparing NDRFT with Repair and DFT

The NDDFT semantics achieves a higher reliability for the system than using the standard DFT semantics [MGN18].Since NDRFT is an extension of the NDDFT, it is expected that the non-deterministic aspect of the semantics ensure that the calculated availability is better than the one with Repairable Dynamic Fault Trees. For this, the results of Repairable Dynamic Fault Tree implementation in [MCD+12] was compared with the obtained results. The paper uses the case study of Active Heat Rejection System (AHRS) to evaluate their semantics. The AHRS case study was used on the NDRFT semantics and compare the availability with their results.

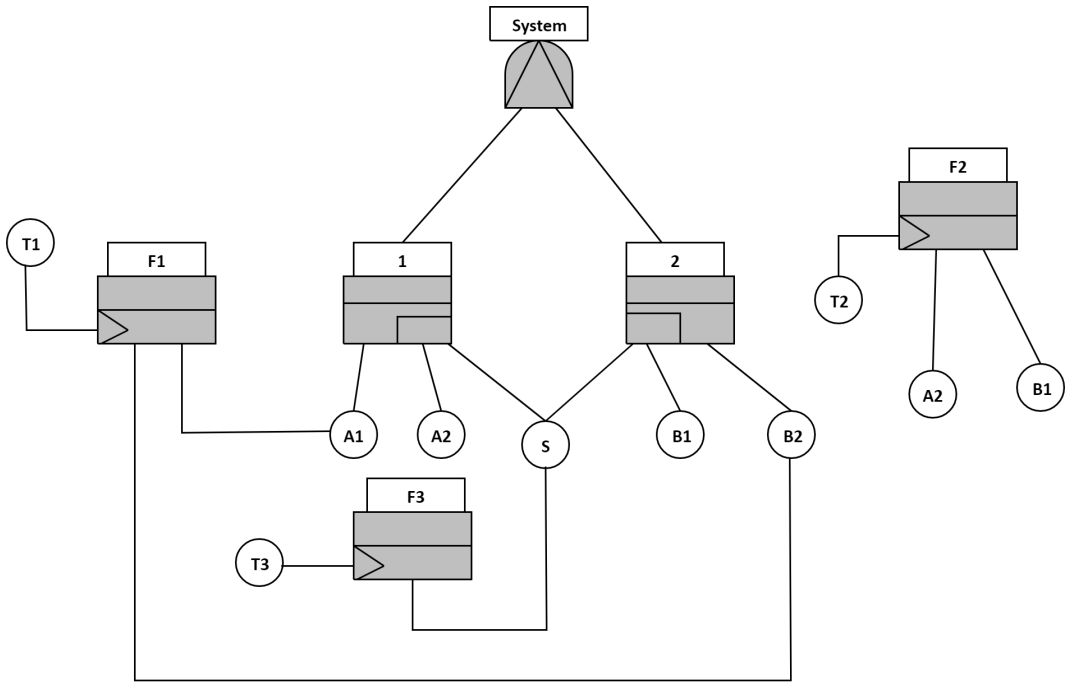


Figure 5.6: Active Heat Rejection System

5.2.4.1 Case Study: Active Heat Rejection System

The Active Heat Rejection System models a system that uses two redundant thermal rejection units 1 and 2. The Fig. 5.6 shows the dynamic fault tree representation of the system. Each of the thermal units have a primary heat rejection unit A1 and B1, spare heat rejection unit A2 and B2 and a common spare heat rejection unit S that is shared by both the redundant units 1 and 2. This use of spare resources is depicted by the use of SPARE gates. All the spare resources are cold spares i.e. they remain in a passive state unless they are activated and used by the spare gate. The common spare unit S is powered separately by power source T3 and is thus functionally dependent on it. Similarly, the units A1 and B2 are powered separately by the power source T1 and units A2 and B1 powered by power source T2. This functional dependency between the thermal units and power source are depicted by the use of FDEP gate as shown in the figure.

5.2.4.2 Results

The availability of the system at T=100 is:

- NDRFT : 99.997 %
- RDFT : 99.891 %

The results show that the non deterministic approach in dynamic repairable fault tree analysis produces slightly better results in terms of availability of the system than the fixed order DFT

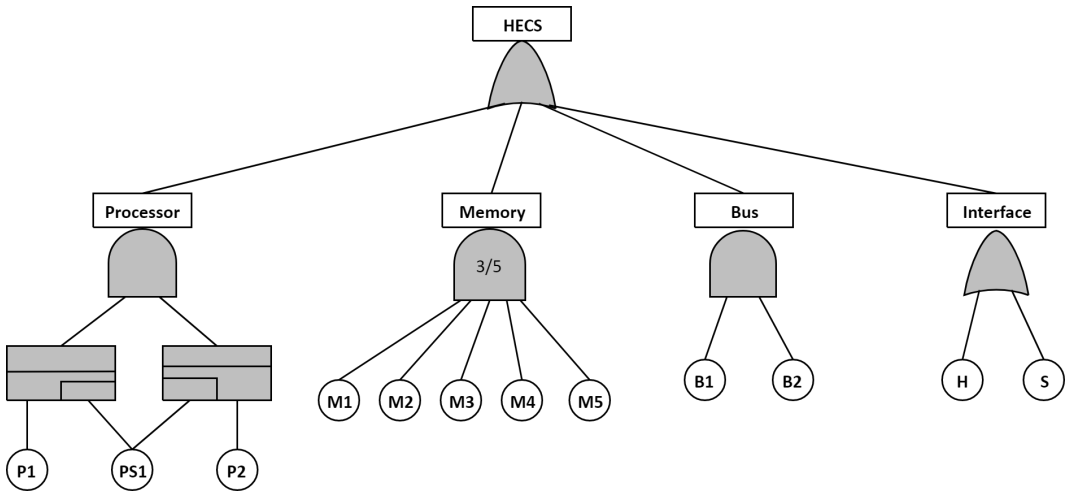


Figure 5.7: Hypothetical Example Computer System

formalism with repair. The flexible order of claiming in NDRFT ensures that a better spare resource is chosen over a bad spare which is not possible with fixed order claiming in DFT.

5.2.5 Evaluating State Space growth and reduction

We expect the state space of the generated Markov automaton and recovery automaton to grow exponentially with the introduction of repair. We evaluate the effect of the new semantics on the generated Markov automaton and Recovery automaton state spaces of each gate. We use a case study to evaluate the increase in the state spaces as well as effectiveness of the reduction algorithms on the generated automatons.

5.2.5.1 Case Study: Hypothetical Example Computer System

We consider the example of Hypothetical example computer system [SVD+02] referred to as HECS, commonly used for analyzing Dynamic Fault trees. The HECS consists of four sub-systems: Processor, Memory, Bus and Interface. Failure of any of its sub-systems will lead to failure of computer system. The processor sub-system uses dual-redundant processors P_1, P_2 and a cold spare processor P_s. This is modeled using a Spare gate with the secondary input connected to the common spare. The memory system interface consists of 5 memory units. The HECS is operational as long as 3 memory units are functioning. K-Vote Gate is used to model this, such that failure of 3 units is required to propagate the failure to system level failure. The Bus System uses two redundant buses B_1, B_2 thus using an AND to model it. The interface sub-system of the HECS can fail either due to hardware failure or software failure which can be conveniently modeled using an OR gate.

There were two experiments conducted using the given case study:

1. Experiment 1: The first experiment was done by gradually increasing the number of

components that are repairable. This was done to study the effect of the number of components on the state space size of both the recovery automata and markov automata and also to evaluate the effect of the minimization algorithm on the recovery automata.

2. Experiment 2: The second experiment using the same case study example by considering different sub-systems of the whole system as repairable. This was done to study the effect of different type of gates with repair components on the overall state space of the generated automata.

5.2.5.2 Results

1. Experiment 1: The Fig. 5.8 shows the increase in the recovery automaton states and transitions with increasing repairable components. The components are made repairable from left to right starting from p1 to S. The results show that the minimization algorithms can reduce the state space of recovery automaton when the system has some of the components repairable. The effect of minimization reduces as the numbers of components considered repairable increases due to the introduction of more number of outgoing transitions from each state.

   The Fig. 5.9 shows a similar increase in state space of the markov automaton states and transitions with increase in components considered repairable. We can see that there is exponential growth of the state space size depending on the number of repairable components.

2. Experiment 2: The Table. reftable:hecsra shows the number of states and transitions of the generated recovery automaton with different sub-systems considered repairable and also the effect of the minimizers on the state space. The Memory sub-system which uses the 3/5 - Vote gate shows a maximum increase in recovery automaton state space size when it is considered repairable. It is followed by the Interface sub-system that uses the AND gate. Reducing the recovery automata using the minimization algorithm reduces the state space for all cases. Among the reduced automata, the Memory sub-system still has a higher number of states and transitions in comparison to other gates due to more number of inputs. The RA state space of the system with Interface repairable is also high.

   The Table. 5.4 shows the number of states and transitions of the generated markov automaton. When the interface sub-system is made repairable, the system generates the markov automaton with more number of states and transitions. This is followed by the Memory Unit which also generates a large state space.

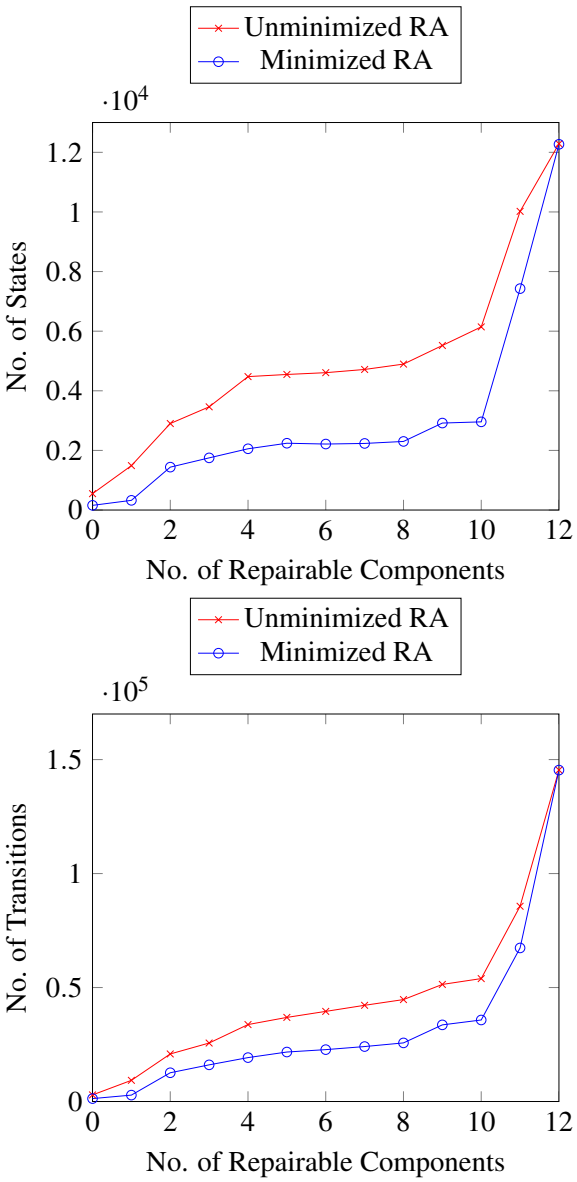


Figure 5.8: Growth and Minimization of Recovery Automata a) States and b) Transitions

Minimization	–	None	Processor	Memory	Bus	Interface
None	States	549	1454	3462	313	3309
	Transition	2893	10991	25655	2564	20235
Trace Equivalence	States	235	289	1795	313	937
	Transition	1767	2784	16303	2564	7068
Adding Orthogonal	States	214	289	1770	299	867
	Transition	1681	2784	16155	2513	6717

Table 5.3: Recovery Automata State Space of HECS System with different sub-systems being repairable

Markov automaton	None	Processor	Memory	Bus	Interface
States	1251	2462	5882	2395	6525
Transitions	5925	16688	31262	13220	35280

Table 5.4: Markov Automata State Space of HECS System with different sub-systems being repairable

Additionally, the execution time for the fault tree analysis process that includes transformation to Markov Automata, synthesizing Recovery Automata, transformation to Markov chain and measurement of system metrics were calculated. Fig. 5.10 shows the increase in execution time with increasing the number of components considered repairable.

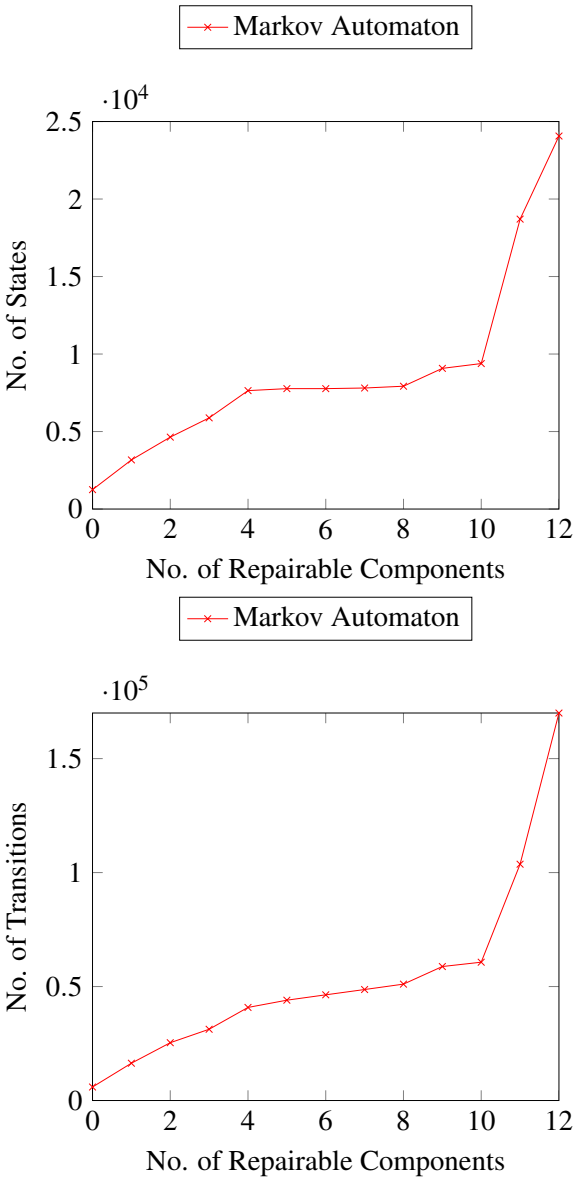


Figure 5.9: Growth of Markov Automata a) States and b) Transitions

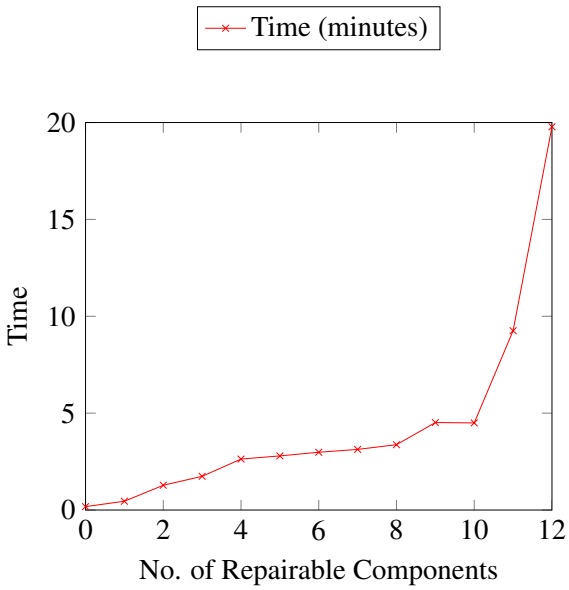


Figure 5.10: Execution Time for Fault Tree Analysis

5.3 Discussion of the Results

The evaluation results show that the semantics developed produces the expected results. From the first three experiments, the implementation of the semantics and correctness of minimization was verified and the third experiment shows the calculated metrics using the given semantics. The gate semantics when compared with the use case study of a similar implementation showed similar results. The Recovery Automaton produced the same results before and after applying the minimization algorithm, thus proving the correctness of the minimization done. Additionally, the different system metrics were analysed for the NDRFT semantics.

The developed semantics also shows improved availability in comparison to RDFT semantics. Since the NDRFT semantic is an extension to NDDFT semantic, the NDRFT also shows better availability of the system due to better management of spare resources. The RDFT on the other hand, maintains the strict ordering of spare resources claiming as is the case for DFTs.

The experiments for evaluating the state space show exponential increase in number of states and transitions of Markov Automaton and Recovery Automaton when the number of components that can be repaired gradually increase in a system. The introduction of the notion of repair and additional recovery action is responsible for the additional increase in states and transitions. The minimization algorithms for the Recovery Automaton were effective in reduction when the system was partially repairable.

Finally, based on the evaluation results, the limitations that were observed were as follows

- **State Space:** For larger systems and systems with more number of repairable compo-

nents, the number of states increases exponentially. The effectiveness of minimization algorithms reduce with the increase in number of components that are repairable, due to more number of outgoing transitions per state.

- **Execution Time:** Exponential increase in execution time with larger systems due to the larger state space. The time for execution includes the entire process of transforming the Fault Tree to different automata, performing reduction and evaluating the system metrics. Program times out in some cases when the fault tree is large.

Despite the limitations, we can still use the NDRFT analysis for smaller systems or considering only the sub-systems of a very large system. Also, in a system typically, not all components can be repaired. Especially, in space applications, where the type of repair we deal with is usually software faults since we have little control after we send the satellites to space, the NDRFT analysis can still be applied to the system.

Chapter 6

Conclusion

This chapter provides a summary of the thesis work and discusses the conclusions drawn from the results. A brief discussion on future work on the developed work is also discussed.

6.1 Summary

In this thesis work, semantics for the Non-Deterministic Repairable Fault Tree has been developed. Initially, the state-of-the-art of Fault Trees was investigated to capture existing work on different fault tree semantics and the variants of the fault tree that have been already researched upon.

The Non-Deterministic Dynamic Fault Tree semantics was extended to include the notion of repair and non-permanent faults. Therein, distinction was made between repair event transitions and failure event transitions. Semantics for each individual gate was investigated. Additional recovery action was added to the SPARE semantics, thus giving it the choice to switch between gates when the defective component has been repaired or restored. The developed semantics also tackles the indirect dependency when the primary input is shared with a PAND gate, where the status of the PAND gate affects the possible recovery actions of the Spare gate. Additionally, the semantics for FDEP gate was defined where the failure of a trigger event causes all its dependent events to fail and repair of the trigger causes only the components forced to fail by the FDEP trigger to be restored to working state. The Markov Automaton generation algorithm was further adapted from NDDFT to use the updated semantics of NDRFT.

In the Recovery Automaton synthesizing process, the new semantics requires that distinctions be made between repair and fault event transitions. The already implemented recovery automata synthesizing algorithm from NDDFT was adapted to include repair transitions in addition to fail transitions. Thereby, the best transitions for non-deterministic choices are chosen based on the calculated estimated time to reach the fail state from the given state.

Additionally, the existing minimization algorithms were investigated and adapted to reduce the recovery automaton generated from NDRFT. With the introduction of repair transitions, the recovery automaton has the problem of state space explosion. However, the recovery automaton can be reduced to a smaller automaton with equivalent recovery behavior. The Partition Refinement Algorithm that uses the concept of trace equivalence and Final State rule that was implemented for NDDFT was reused to reduce the recovery automaton up to some extent. Additionally, the algorithm that uses concept of Orthogonality was adapted by extending the rule for repairable systems and applied for reducing the recovery automaton further.

Finally, the implementations were evaluated using different case studies and JUnit test cases. JUnit test cases were run to check the semantics for each gates and gate combinations. The results show that the generated Markov automaton and recovery automaton is consistent with the defined semantics. The correctness of the implementation was verified with different case studies from literature. The state space growth and the effectiveness of the minimization algorithms was also evaluated. The improvement in the achieved Availability of the system when we use NDRFT in comparison to using the RDFT was evaluated.

6.2 Conclusion

Based on the evaluation and results for the implementation of the proposed semantics, the following conclusions can be drawn:

- The state-of-the-art of Fault Tree Analysis was investigated and different varities of fault trees were compared. Currently, in literature there is no existing work covering repairability with non-deterministic recovery actions and dynamic fault trees.
- The Non-Deterministic Repairable Fault Tree semantics was developed that combines Non-Deterministic Dynamic Fault Tree with the notion of repair. It improves the expressive power of the NDDFT fault tree. as it takes into account the non-permanent faults in the system as well. The non-deterministic approach ensures better utilization of spare resources which results in higher availability of the system in comparison to the standard dynamic fault tree approach. Thereby, higher availability results in lower downtime, reduced costs associated with failure and more robust systems.
- The existing Markov Automaton generation algorithm from NDDFT was extended and successfully adapted to work with NDRFT semantics and repair events.
- The Recovery Automaton model and synthesis techniques from previous work was also investigated and adapted for the case of Markov automaton generated from NDRFT that has both repair and failure transitions.

- The developed semantics increases the state space of the generated recovery automaton exponentially due to the introduction of repair transitions and more number of states. The Trace Equivalence and Final State rule based algorithms for minimizing the recovery automaton was reused to reduce the the space size of recovery automaton. The Orthogonal states based minimization algorithm needed to be modified and adapted to work with NDRFT as the rule was based on the assumption that events can occur only once in a system. The assumption is not true for NDRFT and thus the rule needed to be changed to accommodate events occurring multiple times in a system. The algorithm was then adapted and used to further reduce the recovery automaton.

- The minimization algorithms have lower effectiveness as the system has more number of components that can be repaired. This maybe due to the more number of outgoing transitions occurring per state.

- The goals defined in 1.2 have been realized for the semantics and adapting the algorithms for automaton generation. The minimization algorithms have been adapted successfully as well. However, the goal of reduction of the Recovery Automaton state space was only realized for partially repairable systems.

- Additionally, the Markov automaton also suffers from the problem of state explosion. This also has an effect on the time for analysis as larger state space require longer time for analysis. Further work needs to be done to deal with this problem.

6.3 Future Work

In the future, the recovery automaton properties can be further investigated to reduce the state space, since it still has transitions with empty recovery actions. For larger systems with more number of repairable components, the existing minimization algorithms become less effective and thus it needs further techniques to deal with the large state spaces.

The Markov Automaton also suffers from the problem of state space explosion and increases the computation time. Thereby, the Markov automaton requires minimization to improve the efficiency and time for the fault tree analysis.

We could also further account for the dependencies between different components failure and repair events by allowing a component to have different states of degradation. A failure of one of the redundant components can affect the failure rate of another component in some cases due to increased dependency of the system functioning on the available components. Thus taking this dependency into consideration will help model a system with improvement in accuracy of calculated metrics.

Bibliography

[ABVdB+13] Florian Arnold, Axel Belinfante, Freark Van der Berg, Dennis Guck, and Mariëlle Stoelinga, *Dftcalc: A tool for efficient fault tree analysis*, Computer Safety, Reliability, and Security (Berlin, Heidelberg) (Friedemann Bitsch, Jérémie Guiochet, and Mohamed Kaâniche, eds.), Springer Berlin Heidelberg, 2013, pp. 293–301.

[AHMS19] Marwan Ammar, Ghaith Bany Hamad, Otmane Ait Mohamed, and Yvon Savaria, *Towards an accurate probabilistic modeling and statistical analysis of temporal faults via temporal dynamic fault-trees (tdfts)*, IEEE Access **7** (2019), 29264–29276.

[Ava] *Introduction to repairable systems*, http://reliawiki.com/index.php/Introduction_to_Repairable_Systems, Accessed: 2019-03-11.

[Avi76] A. Aviziens, *Fault-tolerant systems*, IEEE Transactions on Computers **C-25** (1976), no. 12, 1304–1312.

[B+99] Kerstin Buchacker et al., *Combining fault trees and petri nets to model safety-critical systems*, High performance computing, The Society for Computer Simulation International, 1999, pp. 439–444.

[BA78] Sheldon B. Akers, *Binary decision diagrams*, Computers, IEEE Transactions on **C-27** (1978), 509–516.

[BCRFH08] Marco Beccuti, Daniele Codetta-Raiteri, Giuliana Franceschinis, and Serge Haddad, *Non deterministic repairable fault trees for computing optimal repair strategy*, Proceedings of the 3rd International Conference on Performance Evaluation Methodologies and Tools, 2008, p. 56.

[DBB92] Joanne Bechta Dugan, Salvatore J Bavuso, and Mark A Boyd, *Dynamic fault-tree models for fault-tolerant computer systems*, IEEE Transactions on reliability **41** (1992), no. 3, 363–377.

[DJKV17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk, *A storm is coming: A modern probabilistic model checker*, International Conference on Computer Aided Verification, Springer, 2017, pp. 592–600.

[EHZ10] Christian Eisentraut, Holger Hermanns, and Lijun Zhang, *On probabilistic automata in continuous time*, 2010 25th Annual IEEE Symposium on Logic in Computer Science, IEEE, 2010, pp. 342–351.

[Eri99] Clifton A Ericson, *Fault tree analysis*, System Safety Conference, Orlando, Florida, vol. 1, 1999, pp. 1–9.

[GHH+13] Dennis Guck, Hassan Hatefi, Holger Hermanns, Joost-Pieter Katoen, and Mark Timmer, *Modelling, reduction and analysis of markov automata (extended version)*, arXiv preprint arXiv:1305.7050 (2013).

[GKS+14] Dennis Guck, Joost-Pieter Katoen, Mariëlle IA Stoelinga, Ted Luiten, and Judi Romijn, *Smart railroad maintenance engineering with stochastic model checking*, Proc. of RAILWAYS, ser. Civil-Comp Proceedings **104** (2014), 299.

[JGKS16] S. Junges, D. Guck, J. Katoen, and M. Stoelinga, *Uncovering dynamic fault trees*, 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), June 2016, pp. 299–310.

[KGF07] Bernhard Kaiser, Catharina Gramlich, and Marc Förster, *State/event fault trees—a safety analysis model for software-controlled systems*, Reliability Engineering & System Safety **92** (2007), no. 11, 1521–1537.

[Kil73] Gary A Kildall, *A unified approach to global program optimization*, Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM, 1973, pp. 194–206.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker, *PRISM 4.0: Verification of probabilistic real-time systems*, Proc. 23rd International Conference on Computer Aided Verification (CAV'11) (G. Gopalakrishnan and S. Qadeer, eds.), LNCS, vol. 6806, Springer, 2011, pp. 585–591.

[Kri06] Duane Kritzinger, *Aircraft system safety: Military and civil aeronautical applications*, Woodhead Publishing, 2006.

[KZH+11] Joost-Pieter Katoen, Ivan S Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N Jansen, *The ins and outs of the probabilistic model checker mrmc*, Performance evaluation **68** (2011), no. 2, 90–104.

[MCD+12] Gabriele Manno, Ferdinando Chiacchio, D D'Urso, N Trapani, and L Compagno, *Raatss, an extensible matlab® toolbox for the evaluation of repairable dynamic fault trees*, International Conference on Probabilistic Safety Assessment and Management, ESREL, 2012, pp. 1–10.

[MGN18] Sascha Müller, Andreas Gerndt, and Thomas Noll, *Synthesizing failure detection, isolation, and recovery strategies from nondeterministic dynamic fault trees*, Journal of Aerospace Information Systems (2018), 52–60.

[MMGN18] Liana Mikaelyan, Sascha Müller, Andreas Gerndt, and Thomas Noll, *Synthesizing and optimizing fdir recovery strategies from fault trees*, International Workshop on Formal Techniques for Safety-Critical Systems, Springer, 2018, pp. 37–54.

[Rel] *Time-dependent system reliability (analytical)*, http://reliawiki.com/index.php/Time-Dependent_System_Reliability_(Analytical), Accessed: 2019-03-11.

[RFIV04] D. C. Raiteri, G. Franceschinis, M. Iacono, and V. Vittorini, *Repairable fault tree for the automatic evaluation of repair policies*, International Conference on Dependable Systems and Networks, 2004, June 2004, pp. 659–668.

[RS15] Enno Jozef Johannes Ruijters and Mariëlle Ida Antoinette Stoelinga, *Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools*, Computer science review **15-16** (2015), 29–62 (English).

[SD13] F. SalarKaleji and A. Dayyani, *A survey on fault detection, isolation and recovery (fdir) module in satellite onboard software*, 2013 6th International Conference on Recent Advances in Space Technologies (RAST), June 2013, pp. 545–548.

[SDC99] Kevin J Sullivan, Joanne Bechta Dugan, and David Coppit, *The galileo fault tree analysis tool*, Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352), IEEE, 1999, pp. 232–235.

[SVD+02] Michael Stamatelatos, William Vesely, Joanne Dugan, Joseph Fragola, Joseph Minarick, and Jan Railsback, *Fault tree handbook with aerospace applications*, 2002.

[TD04] Zhihua Tang and Joanne Bechta Dugan, *Minimal cut set/sequence generation for dynamic fault trees*, Annual Symposium Reliability and Maintainability, 2004-RAMS, IEEE, 2004, pp. 207–213.

[TFLT83] Hideo Tanaka, LT Fan, FS Lai, and K Toguchi, *Fault-tree analysis by fuzzy probability*, IEEE Transactions on reliability **32** (1983), no. 5, 453–457.

[VGRH81] William E Vesely, Francine F Goldberg, Norman H Roberts, and David F Haasl, *Fault tree handbook*, Tech. report, Nuclear Regulatory Commission Washington DC, 1981.

[VJK16] Matthias Volk, Sebastian Junges, and Joost-Pieter Katoen, *Advancing dynamic fault tree analysis-get succinct state spaces fast and synthesise failure rates*, International Conference on Computer Safety, Reliability, and Security, Springer, 2016, pp. 253–265.

[WF13] Alexandra Wander and Roger Förstner, *Innovative fault detection, isolation and recovery strategies on-board spacecraft: state of the art and research challenges*, Deutsche Gesellschaft für Luft-und Raumfahrt-Lilienthal-Oberth eV, 2013.