

Synthesizing FDIR Recovery Strategies for Space Systems

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Sascha Müller
aus Bad Kreuznach, Deutschland


Berichter: apl. Prof. Dr. Thomas Noll
Prof. Dr. rer. nat. Andreas Gerndt
Prof. Dr. Ir. Dr. h.c. Joost-Pieter Katoen

Tag der mündlichen Prüfung: 12.01.2023

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online
verfügbar.

Synthesizing FDIR Recovery Strategies for Space Systems

Sascha Müller

 orcid.org/0000-0002-1913-1719

Submitted: July, 2023

Abstract

Dynamic Fault Trees (DFTs) are powerful tools to drive the design of fault-tolerant systems. However, semantic pitfalls limit their practical utility for interconnected systems that require complex recovery strategies to maximize their reliability.

This thesis discusses the shortcomings of DFTs in the context of analyzing Fault Detection, Isolation and Recovery (FDIR) concepts with a particular focus on the needs of space systems. To tackle these shortcomings, we introduce an inherently non-deterministic model for DFTs. Deterministic recovery strategies are synthesized by transforming these non-deterministic DFTs into Markov automata that represent all possible choices between recovery actions. From the corresponding scheduler, optimized to maximize a given RAMS (Reliability, Availability, Maintainability and Safety) metric, an optimal recovery strategy can then be derived and represented by a model we call recovery automaton. We discuss dedicated techniques for reducing the state space of this recovery automaton and analyze their soundness and completeness. Moreover, modularized approaches to handle the complexity added by the state-based transformation approach are discussed. Furthermore, we consider the non-deterministic approach in a partially observable setting and propose an approach to lift the model for the fully observable case. We give an implementation of our approach within the Model-Based Systems Engineering (MBSE) framework Virtual Satellite.

Finally, the implementation is evaluated based on the FFORT benchmark. The results show that basic non-deterministic DFTs generally scale well. However, we also found that semantically enriched non-deterministic DFTs employing repair or delayed observability mechanisms pose a challenge.

Zusammenfassung

Dynamic Fault Trees (DFTs) sind wirkungsvolle Werkzeuge um fehlertolerante Systeme zu designen. Allerdings limitieren semantische Fallstricke ihren praktischen Nutzen bei eng verknüpften Systemen, welche komplexe Wiederherstellungsstrategien benötigen um ihre Zuverlässigkeit zu maximieren.

Diese Arbeit diskutiert die Schwächen von DFTs in dem Kontext von Fault Detection, Isolation and Recovery (FDIR) Konzeptanalysen mit einem Fokus auf Raumfahrtsystemen. Als Ansatz führen wir ein nicht-deterministisches DFT Modell ein. Durch eine Transformation in Markov-Automaten werden dann Wiederherstellungsstrategien synthetisiert. Für die Synthese wird hierzu der optimale Scheduler ermittelt, welcher eine gegebene RAMS (Reliability, Availability, Maintainability and Safety) Metrik optimiert, und dieser dann in ein von uns Recovery Automat genanntes Modell weitertransformiert.

Um den Zustandsraum des Recovery Automaten zu reduzieren, entwickeln wir dedizierte Techniken und analysieren diese auf ihre Korrektheit. Außerdem werden modulare Ansätze betrachtet um die Komplexitätszunahme durch den Zustandsraumbasierten Ansatz zu handhaben. Weiterhin untersuchen wir den nicht-deterministischen Ansatz in einem bedingt observierbaren Kontext. Für den vorgeschlagenen Ansatz diskutieren wir eine von uns angefertigte Implementierung innerhalb des Virtual Satellite Frameworks.

Abschließend evaluieren wir die Implementierung mithilfe des FFORT Benchmarks. Die Ergebnisse zeigen, dass nicht-deterministische DFTs im Allgemeinen gut skalieren. Es lässt sich allerdings auch feststellen, dass semantisch erweiterte Versionen, welche zum Beispiel mit Reparaturmechanismen oder bedingter Observierbarkeit ausgestattet sind, eine Herausforderung darstellen.

Acknowledgements

Although your journey as the reader is just beginning, mine as the author of this dissertation has come to a close. And while I was the one to conduct the main research and put it all into a final monolithic thesis, in the end, research is never accomplished in a pure vacuum. Various people have given me invaluable help in finishing the PhD journey and I would like to use this section to thank everyone who supported me in the creation of this thesis. Some may have worked directly together with me and authored some papers, others may have helped me by contributing through meaningful discussions. First of all, I would like to extend my gratitude to my two supervisors Thomas Noll and Andreas Gerndt, who have patiently guided me along this long path. Then there are of course various students without whom I would probably be working on this project for more years to come. Thank you Andrey Larpin, Mikaelyan Liana, Adeline Jordon, Alexandros Khan, and Yogeswari Renganathan for helping me push my research forward! Finally, I would like to thank some of my colleagues for their helpful discussions. Thank you Matthias Volk and Kilian Höflinger, your expertise on fault trees and FDIR came to to good use in this dissertation! Thank you Philip Fischer, for giving me various crazy ideas. And last but not least, thank you all dear conference and journal reviewers who provided useful feedback to my papers and my research!

Contents

Acronyms	xiii
1 Introduction	1
1.1 Publication List	2
1.2 Thesis Structure	4
2 State of the Art	5
2.1 FDIR	5
2.1.1 Modular Hardware Redundancy	7
2.1.2 Spare Hardware Redundancy	7
2.1.3 Analytical Redundancy	8
2.1.4 Recovery Strategies for Space Systems	10
2.1.5 Hierarchical FDIR	11
2.2 Robustness Measures	12
2.3 Techniques For Fault Modeling and Analysis	12
2.3.1 Low-level Techniques	14
2.3.2 Classical Techniques	16
2.3.3 Model-Based Techniques	26
2.4 From Fault Model to Recovery Strategy	28
2.5 Partial Observability	29
3 Preliminaries	31
3.1 Basic Notation	31
3.2 Markovian Structures	32

3.2.1	Markov Chains	32
3.2.2	Markov Automata	33
3.3	Dynamic Fault Trees	34
4	Formalization of the FDIR Model	37
4.1	Rate Dependency Extension	41
4.2	Non-Deterministic Fault Trees	42
4.3	NdDFT with Repair	47
4.3.1	FDEP with Repair	49
4.3.2	Extended Notation with Repair	50
4.4	Markov Automaton Semantics	51
4.4.1	Construction Examples	54
4.4.2	Repairable NdDFT to MA	58
4.4.3	Recovery Strategies and Automata	59
5	Synthesis of Recovery Strategies	65
5.1	Synthesis Methodology	65
5.1.1	Extraction	66
5.2	Examples	67
5.2.1	Construction of an Adaptable Recovery Strategy	67
5.2.2	Optimized Spare Ordering	71
5.3	Further Optimization of Recovery Automata	73
5.3.1	Optimizing Orthogonal States	76
5.3.2	Optimizing the FAIL State	81
5.3.3	Completeness	84
5.4	Modular Synthesis of Recovery Automata	91
5.4.1	Modular Workflow	92
5.4.2	Modularization	93
6	Partial Observability	97
6.1	Partially Observable Dynamic Fault Trees	98
6.1.1	MONITOR Gate	99
6.1.2	Gate and Event Semantics	100
6.2	Belief Markov Automaton Semantics	102
6.3	Partially Observable Recovery Automaton	111
6.4	Synthesis Workflow	114
6.4.1	PORA Extraction	114
6.4.2	PORA and MA Synchronization	115
6.4.3	Orthogonality under Partial Observability	116

6.4.4	Adapting Modularization	117
6.5	Synthesis Examples	119
6.5.1	Probabilistic Claim Success	119
6.5.2	Delayed Monitor	120
6.5.3	Failable Monitor	122
6.5.4	Timeout Transitions	124
7	Implementation	129
7.1	Virtual Satellite 4 Framework	129
7.2	Generic Systems Engineering Language	131
7.3	Virtual Satellite 4 FDIR	132
7.3.1	FDIR Conceptual Data Model	134
7.3.2	Analysis CDM	137
7.3.3	Configuration Control	137
7.3.4	Software Workflow for Synthesis	139
7.4	Implementation Details	139
7.4.1	Preprocessing	140
7.4.2	Representation of DFT states	140
7.4.3	Canonical States	141
7.4.4	Optimization Workflow	141
7.4.5	Reducing the Number of Timeout States	142
7.4.6	Selecting Optimal Transitions on the Fly	142
8	Evaluation	145
8.1	Experiment Setup	145
8.2	Fully Observable Scalability Experiments	147
8.2.1	NdDFT Experiments	148
8.2.2	Repairable NdDFT Experiments	150
8.3	Recovery-Equivalence-Based State Space Reduction Experiments	154
8.4	Partially Observable Scalability Experiments	157
8.4.1	Configuration: PO	158
8.4.2	Configuration: PO Delay	161
8.4.3	Configuration: PO Repair	164
8.4.4	Configuration: PO Delay Repair	167
8.5	Summary and Update of the NdDFT Hierarchy	170
9	Conclusion and Outlook	173
	Bibliography	179

Acronyms

BDD	Binary Decision Diagram.	20
BE	Basic Event.	18
BMA	Belief Markov Automaton.	102
BN	Bayesian (Belief) Network.	16
CDM	Conceptual Data Model.	129
CTMC	Continuous Time Markov Chain.	14
DDN	Dynamic Decision Networks.	29
DFT	Dynamic Fault Tree.	1
DRBD	Dynamic RBD.	24
DTMC	Discrete Time Markov Chain.	14
ESA	European Space Agency.	25
FDD	Fault Detection and Diagnosis.	6
FDEP	Functional Dependency.	21
FDI	Fault Detection and Isolation.	8
FDIR	Fault Detection, Isolation and Recovery.	1

FFORT Fault tree FOResT. 145

FMEA Failure Modes and Effects Analysis. 17

FMECA Failure Modes, Effects, and Criticality Analysis. 17

FPG Fault Propagation Graphs. 25

FPTC Fault Propagation and Transformation Calculus. 27

FPTN Fault Propagation and Transformation Notation. 24

FT Fault Tree. 1

FTA Fault Tree Analysis. 1

GSEL Generic Systems Engineering Language. 129

HAZOP Hazard and Operability Studies. 26

I/O-IMC Input-Output Interactive Markov Chain. 28

IM Inspection Module. 30

MA Markov Automaton. 2

MC Markov Chain. 14

MCS Minimum Cut Set. 12

MDB Model-Based Diagnosis. 26

MDP Markov Decision Process. 15

MTTF Mean Time To Failure. 12

NdDFT Non-deterministic Dynamic Fault Tree. 2

PAND Priority AND. 20

PODFT Partially Observable Dynamic Fault Tree. 98

POMA Partially Observable Markov Automaton. 30

POMDP Partially Observable Markov Decision Processes. 30

POR Priority OR. 20

- PORA** Partially Observable Recovery Automaton. 113
- PRA** Probability Risk Assessment. 24
- RA** Recovery Automaton. 2
- RAMS** Reliability, Availability, Maintainability and Safety. 2
- RBD** Reliability Block Diagram. 23
- RFT** Repairable Fault Tree. 23
- SFT** Static Fault Tree. 18
- SHARD** Software Hazard Analysis and Resolution in Design. 26
- SPN** Stochastic Petri Net. 15
- SSA** Steady State Availability. 12
- TFPG** Timed FPG. 25
- TLE** Top-Level Event. 18
- TMR** Triple Modular Redundancy. 7
- VirSat** Virtual Satellite 4. 129
- x-RDEP** Rate Dependency. 41

Chapter 1

Introduction

Reliability engineering is an important discipline in the design of any safety critical system, in particular in the domain of aerospace systems and spacecraft. No matter how well designed a system is, it still has to deal with the presence of faults to some extent. Faults in this context can be events such as equipment failures, wrong sensor readings, external interferences and many more. To raise trust in handling system failures, reliability engineering tries to embed Fault Detection, Isolation and Recovery (FDIR) concepts. These concepts are derived using various tools and methodologies such as Fault Tree Analysis (FTA) [1].

FTA is a methodology commonly used in the industry for performing state-of-the-art failure analysis [2]. The resulting Fault Trees (FTs) describe how faults propagate through components and subsystems of a system and eventually lead to a top-level system failure. Graphical representations of these trees are intuitive and easy to understand. On the one hand, FTs can be used to analyze the system qualitatively in terms of fault combinations that lead to system failure. On the other hand, they also enable quantitative analysis of important computable measures such as reliability. Dynamic Fault Trees (DFTs) are an extension introducing temporal understanding and new features to analyze redundancy concepts known as spare management. However, there are challenges arising from non-deterministic DFT behavior such as spare races. An example for such race behavior can be seen in a system of two operative memories together with a pool of two spare memories. In case both operative memories fail simultaneously it is unclear which backup memory takes over the role of which operational one.

To overcome this shortcoming, we present a new methodology in this thesis. We introduce a model of Non-deterministic Dynamic Fault Trees (NdDFTs) as an extension to DFTs. In contrast to the latter, the new NdDFT does not impose a fixed, rigid order on the spares to be used. As next step, the methodology foresees transforming this NdDFT model into a Markov Automaton (MA) which is suitable for the computation of the aforementioned non-deterministic decisions on spare activations. By optimizing the scheduling of the MA model in terms of a given Reliability, Availability, Maintainability and Safety (RAMS) metric, a deterministic recovery strategy for the NdDFT can be synthesized. This recovery strategy is represented by an object we call a Recovery Automaton (RA), and defines which spare has to be used in which failure state of the system.

1.1 Publication List

The following gives a list of publications created during the thesis creation. Related publications contain direct contributions to the thesis as described in the following Section 1.2. Further publications may have ties to the research discussed in this thesis, but are generally not directly related.

Related publications

1. Sascha Müller, Andreas Gerndt, and Thomas Noll. Synthesizing FDIR recovery strategies from non-deterministic dynamic fault trees. In *2017 AIAA SPACE Forum*, volume AIAA 2017-5163. American Institute of Aeronautics and Astronautics, 2017. doi:[10.2514/6.2017-5163](https://doi.org/10.2514/6.2017-5163)
2. Sascha Müller and Andreas Gerndt. Towards a conceptual data model for fault detection, isolation and recovery in Virtual Satellite. In *SECESA 2018*. European Space Agency, 2018. URL: <https://elib.dlr.de/122061/>
3. Liana Mikaelyan, Sascha Müller, Andreas Gerndt, and Thomas Noll. Synthesizing and optimizing FDIR recovery strategies from fault trees. In *International Workshop on Formal Techniques for Safety-Critical Systems*, pages 37–54. Springer, 2018. doi:https://doi.org/10.1007/978-3-030-12988-0_3
4. Sascha Müller, Andreas Gerndt, and Thomas Noll. Synthesizing failure detection, isolation, and recovery strategies from nondeterministic dynamic fault trees. *Journal of Aerospace Information Systems*, 16(2):52–60, 2019. doi:<https://doi.org/10.2514/1.I010669>

5. Sascha Müller, Liana Mikaelyan, Andreas Gerndt, and Thomas Noll. Synthesizing and optimizing FDIR recovery strategies from fault trees. *Science of Computer Programming*, 196:102478, 2020. doi:<https://doi.org/10.1016/j.scico.2020.102478>
6. Sascha Müller, Adeline Jordon, Andreas Gerndt, and Thomas Noll. A modular approach to non-deterministic dynamic fault trees. In *International Conference on Computer Safety, Reliability, and Security*, pages 243–257. Springer, 2021. doi:[10.1007/978-3-030-83903-1_16](https://doi.org/10.1007/978-3-030-83903-1_16)

Further Publications

1. Kilian Höflinger, Sascha Müller, Ting Peng, Moritz Ulmer, Daniel Lüdtke, and Andreas Gerndt. Dynamic fault tree analysis for a distributed onboard computer. In *2019 IEEE Aerospace Conference*, pages 1–13, 2019. doi:[10.1109/AERO.2019.8742128](https://doi.org/10.1109/AERO.2019.8742128)
2. Sascha Müller, Kilian Höflinger, Michal Smisek, and Andreas Gerndt. Towards an FDIR software fault tree library for onboard computers. In *2020 IEEE Aerospace Conference*, pages 1–10, 2020. doi:[10.1109/AERO47225.2020.9172756](https://doi.org/10.1109/AERO47225.2020.9172756)
3. Emanuel Kopp, Sascha Mueller, Fabian Greif, and Anko Boerner. Towards an H/W-S/W interface description for a comprehensive space systems simulation environment. In *2020 IEEE Aerospace Conference*, pages 1–14, 2020. doi:[10.1109/AERO47225.2020.9172440](https://doi.org/10.1109/AERO47225.2020.9172440)
4. Philipp M Fischer, Caroline Lange, Volker Maiwald, Sascha Müller, Andrii Kovalov, Janis Häseker, Thomas Gärtner, and Andreas Gerndt. Spacecraft interface management in concurrent engineering sessions. In *International Conference on Cooperative Design, Visualization and Engineering*, pages 54–63. Springer, 2019. doi:[10.1007/978-3-030-30949-7](https://doi.org/10.1007/978-3-030-30949-7)

Supervised Theses

1. Yogeswari Renganathan. Semantics of non-deterministic repairable fault trees. Master’s thesis, Technische Universität Darmstadt, 2019. URL: <https://elib.dlr.de/131219/>

1.2 Thesis Structure

The following section gives an overview of the structure of the thesis. For each chapter, we also describe contributions by third parties such as supervised students and give references to prior publications where the results have been pre-published.

Chapter 2 first gives an overview over the topic of FDIR, related analysis methods, metrics, and formal mathematical models. The focus is on presenting a qualitative overview without mathematical details. The chapter received contributions by the student Andry Larpin, who created a survey over existing fault tree dialects and the expressive power of their gates.

Chapter 3 then focuses on formalisms employed in the main chapters and defines the necessary notation and formal definitions.

Chapter 4 presents the basic methodology by defining a formal NdDFT model in terms of Markov Automaton semantics. The chapter is based on the paper [3] and the later formalism refinement published in the journal [6]. Section 4.3 has contributions from the master thesis by Yogeswari Renganathan [13].

Chapter 5 describes the core methodology for synthesizing recovery automata from NdDFTs. The chapter is also partially based on the papers [3, 6]. Section 5.3 is based on the results of the supervised student Liana Mikaelyan and published in [5]. Section 5.4 is based on the results from the supervised student Adeline Jordon and has been partially published in [8].

Chapter 7 gives details on the implementation and describes the integration of the methodology into the Virtual Satellite 4 framework. The FDIR conceptual data model was partially presented in the paper [4].

Chapter 8 investigates the applicability of the synthesis methodology by evaluating various NdDFT classes on the basis of the FFORT benchmark set. Some of the results related to the evaluation of the modularization approach were published in [8].

Chapter 9 gives a summary of the achieved results and outlines avenues for future research.

Chapter 2

State of the Art

This chapter focuses on giving an overview of the state of the art regarding FDIR. We discuss the core concepts that make up the operational procedure known as FDIR and which mechanisms and strategies are commonly employed. Following that, the analysis methods used to judge the quality of an FDIR concept, their connection to formal methods, and analysis metrics are discussed. The main focus lies on introducing important qualitative concepts and terms while leaving the formal definitions this work builds up upon to the follow-up Chapter 3.

2.1 FDIR

To discuss the concepts behind FDIR, we first have to fix the understanding of the word *fault* and how it relates to other terms that all describe that a component or system is exhibiting abnormal behavior. To describe the different classes of abnormal behavior, consequences, and causes, the classification introduced in [14] will be applied.

While a *fault* abstractly describes the notion of some anomaly having occurred, a *failure*, or in some literature also introduced as *hazard*, is a loss of system performance ultimately caused by the occurrence of faults. A component failure may thus be nothing more than a fault when viewed on a system-wide level. Examples for faults include sensor malfunctioning, loss of some equipment, memory corruption, and mechanical malfunctions. Finally, an *error* is a concrete, physical manifestation of a fault.

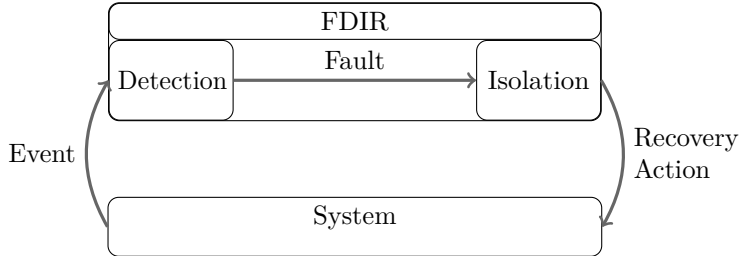


Fig. 2.1. System-FDIR interaction. The FDIR observes events produced by the system and if an event is identified as a fault, a corresponding recovery action is generated.

The goal of any FDIR system lies in the prevention of failure occurrence. While many different strategies and techniques exist that aim to achieve this result, most share a common procedural approach [15]:

1. Monitor the system to detect the occurrence of faults through observed errors.
2. Identify the fault and localize it within the system.
3. Isolate the fault and prevent further propagation of the fault into other parts of the system.
4. Perform counter-measures to recover the system and return it to a stable state.

Some works also consider the notification of the ground segment as a central function of FDIR [16]. Fig. 2.1 illustrates the interaction between the main system and the FDIR.

There exists a wide range of FDIR strategies for creating fault-tolerant systems. They can be mainly classified into two different types [17]: Strategies for Fault Detection and Diagnosis (FDD) and Recovery Strategies or Reconfiguration Strategies. The former deals with the first three steps of FDIR and the latter is dedicated to handling the recovery aspect. While recovery strategies are often kept relatively simple [16], sophisticated methods for fault detection and localization have been developed over the last decades, as will be shown in the following sections. As will be seen in the following, redundancies play an integral role in performing FDD and recovery. These redundancies can be introduced in two forms: In the simplest case as physical redundancies in the form of

real, redundant hardware. Alternatively, they can be introduced as analytical redundancies that exploit mathematical relations between the signals to detect possible anomalies and possibly even substitute a failed sensor.

Furthermore, a more advanced notion of *Health Management Systems*, that not only reactively monitor and recover failures but also attempt to avoid failures preemptively, aiming to further increase the system dependability and also its performance, has been developed [18]. Also in the domain of spacecraft, effort has been put into hierarchically classifying faults to deal with them on a more finely grained basis. Section 2.1.5 shows an example of such a hierarchy.

2.1.1 Modular Hardware Redundancy

The first step to FDIR lies in discovering faults and determining whether a unit or some signal created by a unit is faulty. This can be achieved by introducing the notion of *voting mechanisms* or also referred to as *modular redundancy*. Having multiple redundant units employed in parallel gives rise to the question: Which units actually computed the correct result? Initially introduced in [19] by Von Neumann, voting mechanisms aim to tackle this problem by having the units send their outputs not directly to the following computation unit but to an intermediary voter instead. The voter then decides which result to forward using one of the popular voting schemes such as *majority voting* which forwards the most common input and generates an alarm for the incorrect voting inputs. Since the alarm reveals the faulty units, modular redundancy can be used to perform FDD.

Popular in industry practice is Triple Modular Redundancy (TMR). TMR uses a majority voter, a configuration capable of dealing with single point unit failures and extensively discussed in [20]. System-wide fault-tolerance can then be drastically increased by chaining multiple such redundancies plus voting units as depicted in Fig. 2.2 on the next page.

Moreover, multiple unit failure can also be handled by employing more redundancies. In the case of majority voting, in order to obtain a majority of the correct values in the presence of f faults, $2f + 1$ redundancies are necessary.

2.1.2 Spare Hardware Redundancy

The standard for FDIR practice within the industry is the employment of hardware redundancies in the form of *spares*. The use of duplicate hardware provides spare units, which can be used if the currently used operational unit is considered unrecoverable using other means. This approach has proven itself as

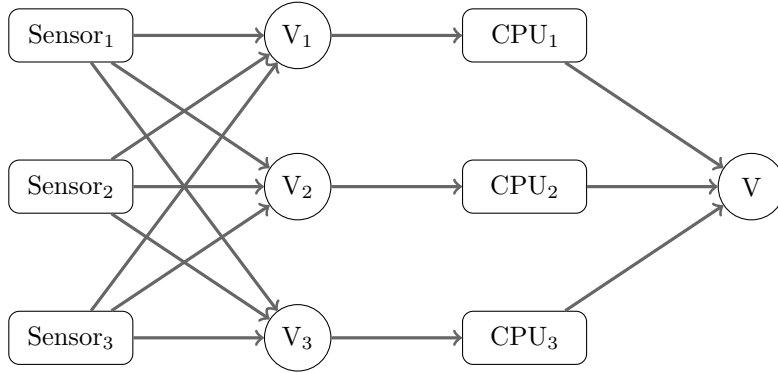


Fig. 2.2. Example sequence of units employing TMR. TMR is applied on the sensor level and then again on the CPU level.

robust and well-performing [21]. Since spare units can be used to recover the system from the failure of individual units, strategies deciding when to activate a spare are recovery strategies. These spares are further refined into three different types:

- **Hot redundant:** Hot redundant units are active, just like the primary unit. If the primary unit becomes faulty, switching to the hot redundant unit is nearly instantaneous since no further boot-up sequence is needed.
- **Cold redundant:** Cold redundant units are dormant, and to activate them in the case of a fault, they need to be booted up. This may make the system vulnerable for the duration of the boot process.
- **Warm redundant:** Warm redundant units are a compromise between hot and cold redundant. The unit is in a booted state but not actively used. An example of such a state would be a standby state, which allows for quick switching with a shorter boot-up time while not risking faster unit degradation.

2.1.3 Analytical Redundancy

While modular hardware redundancies provide a high level of robustness, it is also resource-intensive and sometimes simply not feasible due to increased required mass and volume. *Analytical redundancies*, also called Fault Detection

and Isolation (FDI), circumvent this issue by introducing redundancy using mathematical models and estimation techniques, only requiring computational resources. FDI forms a significant part of FDD, in fact so much that in some literature, they are equated to being the same [17]. However, more recent approaches also employ analytical redundancies outside of detection and isolation to recover from lost sensor hardware [22]. The core approach for analytical redundancy is displayed in Fig. 2.3.

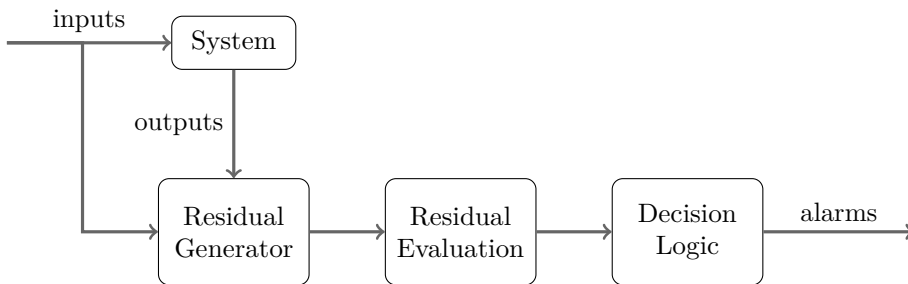


Fig. 2.3. FDI unit using Analytical Redundancy. The expected output and the real system output are compared to detect anomalies.

Here, additional knowledge of the system is leveraged, effectively introducing another layer of redundancy for detecting abnormal behavior without introducing new physical redundancies. For example, by measuring the current produced by a solar array, which implies a certain orientation of a spacecraft towards the sun, the expected outputs for a sun sensor can be restricted. In general, such an FDIR system consists of the following main units [17]:

1. A Residual Generator computes the difference (residual) between the measured feature and its expected value.
2. A Residual Evaluation determines whether a residual is considered large enough to be an anomaly.
3. A Decision Logic performs the fault localization and diagnostic logic, finally raising the necessary alarms to inform the system about a fault.

Kalman filters which attempt to estimate the next time step value of a time-evolving variable while considering possible noise interference [23] can be seen as a very early approach to such an analytical redundancy by using statistical methods.

However, while Model-based FDI has gathered quite some interest on the scientific side - and some techniques such as Kalman filters have indeed impacted modern industry practices - the approach itself has not received as much positive reception in real-life industry applications. The authors of [24] credit this to the lack of accuracy of Model-based FDI as many models heavily rely on simplifications (such as assuming time linear kinematic models). A survey of applied model-based FDIR, in particular in the area of aerospace, can be found in [17].

2.1.4 Recovery Strategies for Space Systems

A first class of recovery strategies in the form of redundancy concepts has already been covered in the previous section. In this section, further recovery strategies that have emerged in the domain of space systems are discussed. Relevant vocabulary for classifying recovery strategies are the notions of automatic and autonomous processes. In [25] *automatic processes* are replacements of manual operations that may still include human participation in addition to the automatic execution of software or hardware procedures. *Autonomous processes* on the other hand, are independent from any form of human interaction.

Half-Satellite Strategy A very conservative strategy that heavily depends on human interaction but strongly guarantees the safety of the spacecraft is reported in [16] as a Half-Satellite strategy. Here, whenever any fault is detected, a complete reconfiguration is performed by switching all units to a redundancy, and then the spacecraft waits for ground intervention. Since all units are switched to a redundancy, this strategy guarantees that the faulty units are replaced, even when lacking knowledge of the concrete type of the fault or further diagnosis information. This strategy was also reported to be easy to verify due to its simplicity.

Safe Mode A Safe Mode, also sometimes referred to as Survival mode, is a system mode focusing on the spacecraft's survival. The satellite is set to sun pointing and all non-essential units (e.g., payload) are powered off to reduce power consumption and to reduce the risk of interference with survival critical units [26]. The spacecraft is expected to survive ideally for an indefinite amount of time, or at least for a time period significantly longer than the expected ground reaction time [27]. Usually, it is required that a spacecraft can reach the Safe Mode autonomously [28].

2.1.5 Hierarchical FDIR

Different faults may have varying impacts on the system performance and thus may require more drastic or more immediate measures, such as switching to redundant units or, in the extreme case, even relying on ground support intervention. Meanwhile, faults that do not critically affect the system performance should be recovered using less drastic recovery mechanisms such as reboots or re-initializations. A strategy to deal with these different fault classes is the categorization of faults into different impact levels. A fault on a lower level is considered less critical and is usually also more likely to occur. If the FDIR system cannot recover a unit on a lower level, the higher levels are notified via alarms. These may then in return recover the fault using more powerful recovery actions or continue to propagate the alarm to even higher levels. While different hierarchy structures with varying degrees of detail exist for different systems and application purposes, in the context of space systems, a rough hierarchical categorization has been obtained in [15].

Level	Faults	Impact	Fault Detection	Recovery
0	Unit internal	No impact on system performance	Consistency checks, Data transmission checks	Retry, reboot, reinitialize unit
1	Subsystem Software Examples: Subsystem intercommunication failure, Subsystem equipment failure	Subsystem performance degraded	Limit checking of unit parameters, Plausibility checks	Switch to redundant unit, retry, reboot
2	System Reconfiguration Examples: Subsystem failures not recoverable in level 1, Power failures	Subsystem performance loss	Alarms from lower levels Consistency Checks	Platform level switch to redundancy, retry
3	System Control Software Examples: FDIR unit failure	Subsystem performance loss	FDIR Alarms	Platform level switch to redundancy, retry
4	Flight Operations System on Ground Examples: Major overall system failure, deployment failure	System performance loss, mission interruption	Multiple alarms from lower levels, Hardware alarms	Enter Safe mode and wait for ground intervention

Table 2.1: FDIR hierarchical levels for space systems.

While detection and recovery receive much attention in the FDIR design of spacecraft, isolation is often at most performed at the unit level. Recovery means of the different hierarchical impact levels should only be conducted if the higher levels' recovery actions to handle the fault are not hindered.

2.2 Robustness Measures

Before diving into the actual techniques for Failure Analysis, we introduce some notable measures that are interesting to obtain from the analysis. In particular, we can distinguish two classes of such measures, namely *qualitative* and *quantitative* measures.

Qualitative measures provide insight into the structure of fault relationships. For example Minimum Cut Sets Minimum Cut Set (MCS) describe minimum sets of faults whose occurrence lead to a system failure. On the side of quantitative measures, we note four remarkable measures that can be found in many reliability engineering-related literature [29, 30, 31]:

- **Reliability after time t :** Reliability describes the probability that a system is still functional up to time t . In the literature, the reader will find it often denoted by $R(t)$. Failure models from which we can compute reliability measures can answer questions such as “Is the probability of experiencing no failure in a 10 year mission greater than 90%?”
- **Mean Time To Failure:** The Mean Time To Failure (MTTF) describes the period of time that will pass on average until the first system-level failure has occurred. Naturally, this measure aims to provide the expected lifetime of the system. For repairable systems, where $R(t)$ does not converge towards 0, the MTTF is not a useful measure as it becomes infinite.
- **Availability after time t :** This measure describes the fraction of time the system is functional after time t and is often denoted by $A(t)$. In particular, in repairable systems, the notion of availability is of high interest since it can capture whether or not the system recovery mechanisms are effective enough to deal with system-level failures.
- **Steady State Availability:** The Steady State Availability (SSA) describes the long-term availability of a system. It corresponds to the converged value of $A(t)$ for $t \rightarrow \infty$.

2.3 Techniques For Fault Modeling and Analysis

In order to construct effective FDIR systems, a clear understanding of the fault behavior of the system under investigation is required. To this aim, several approaches to analyzing fault behavior have been proposed and have already

2.3. TECHNIQUES FOR FAULT MODELING AND ANALYSIS

become deeply rooted in the process of safety and reliability engineering. Upon review, we have derived that the techniques can be mostly categorized into the following classes:

- **Low-Level Techniques:** We list here techniques for creating fault models that involve the usage of very general-purpose formalisms, such as Markov chains. These low-level techniques yield models that can be difficult to comprehend and create by an engineer but are easy to evaluate due to significant preexisting research.
- **Classical Techniques:** Here, classical failure analysis techniques are listed, which require an engineer to manually perform a failure analysis given an existing system model. An example of a classical technique is the Fault Tree Analysis. To evaluate the models resulting from these techniques, they are first transformed into a lower-level fault model, from which the measures of interest are then computed.
- **Model-Based Techniques:** Model-based techniques abstract away the manual creation of the full fault model. Here, approaches are listed which allow generating a classic fault model (such as a fault tree) from given system information and a simple basic fault model. It should be noted that Model-Based Failure Analysis is not related to Model-Based FDD.

Fig. 2.4 gives an overview of the landscape of failure analysis techniques, which will be introduced in the following part.

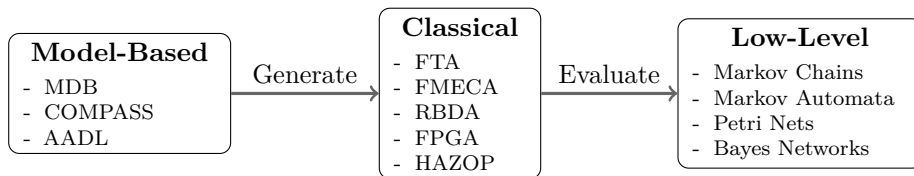


Fig. 2.4. Landscape of failure analysis techniques. From general models describing system architectures, classical fault models are derived and evaluated using low-level mathematical models.

2.3.1 Low-level Techniques

In this section, low-level techniques for fault modeling will be considered. While they are not practical to use by hand, these low-level techniques can employ a plethora of background theory and mathematical tools dedicated to evaluate them. In the following sections, it will be seen that higher-level models are often handled by transforming them into lower-level models such as the ones reviewed in this section.

2.3.1.1 Markov Chains

Markov Chains (MCs) [32] are *state-based* systems with *probabilistic transitions*. In contrast to standard finite automata, where the successor state is chosen based on an input symbol, MCs choose the successor of a state based on a probability distribution. There exist two main categories of MCs based on the handling of time:

- **Discrete Time Markov Chains (DTMCs):** Here time is discretized. In each discrete time step, a successor is chosen. To model this each transition is equipped with a transition probability. The sum of all outgoing transition probabilities has to sum to unity. Only one transition can be taken at each time step.
- **Continuous Time Markov Chains (CTMCs):** In this model, time is considered a continuous object. Within some time step, any arbitrary amount of transitions can be taken. Since an arbitrary number of fired transitions is allowed, the special case of no transition firing at all is also possible. Self-loops for modeling remaining in a state are therefore not necessary. Here, each transition is equipped with a transition rate dictating the rate of transitioning to a successor state.

The following example illustrated in Fig. 2.5 on the following page showcases how a failure model for the case of “Fail if faults A and B have occurred” would be expressed in the Markov formalism.

If starting in s_0 and considering s_3 as the failed state, the faults A and B have to occur for it to be reached. MCs do not possess a unique initial state; instead, they can be equipped with an initial probability distribution. In this example, the initial probability distribution would be $P_0(s_0) = 1$ and $P_0(s) = 0$ for any other state $s \in \{s_1, s_2, s_3\}$. By answering the question

“What is the probability of reaching some state s after some time t ?”

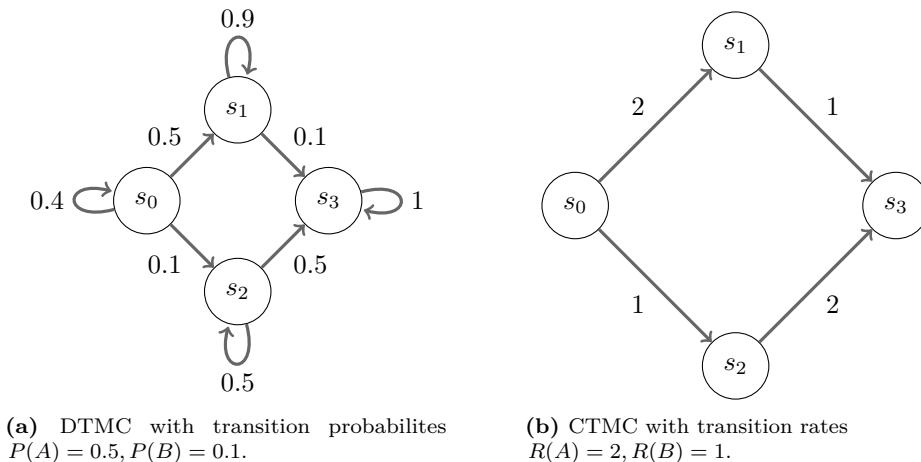


Fig. 2.5. Example DTMC and analog CTMC modeling “A and B”.

interesting properties for failure analysis, such as reliability, can be computed. In the above example, s_3 models the failed state where both A and B have occurred. Computing the reliability of the system after time t thus reduces to finding the answer to the question “What is the probability of reaching s_3 after time t ?”.

2.3.1.2 Markov Automata

MCs have also been extended to incorporate immediate, non-deterministic transitions. A Markov Decision Process (MDP) [32] is a structure extending DTMCs in such a manner. For CTMCs, the model of Markov automata, which additionally also extends the model with probabilistic transitions, has been introduced in [33].

2.3.1.3 Petri Nets

Petri nets are graph-based structures for modeling concurrent processes, with Stochastic Petri Nets (SPNs) being their stochastic generalization for the use of concurrent, probabilistic processes [34]. Due to fault occurrence’s probabilistic and concurrent nature, SPNs are well suited for fault modeling. A Petri net contains two types of nodes:

- **Places** which can contain an unconstrained number of tokens and
- **Transitions** which take one token from all incoming places and then put one token on all outgoing places. In SPNs, each transition t is additionally associated with a probability $P(t)$ that determines that t fires during a discrete time step.

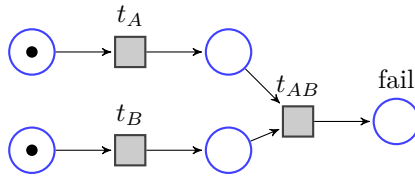


Fig. 2.6. Example Petri net modeling “A and B”. In an SPN, the transitions t_A , t_B may also be equipped with probability each.

An example Petri net is given in Fig. 2.6. Setting $P(t_A) = 0.5$, $P(t_B) = 0.1$ and $P(t_{AB}) = 1$ yields an SPN that produces the same reliability as the DTMC in Fig. 2.5a.

2.3.1.4 Bayesian Networks

A Bayesian (Belief) Network (BN) is yet again a graph-like structure that relates observable variables (e.g., sensor values, watchdog alarms) to hidden variables (e.g., whether a given sensor is faulty) [35, 36]. Each node in the network represents a Boolean variable associated with a probability table conditioned on the parent variables. A Bayesian network can then be fed with observed evidence, and queries such as “What is the probability of fault A having occurred?” can then be posed to the BN. In this sense, BNs focus on providing diagnostic capabilities instead of the propagation-focused view underlying MC and SPN models.

2.3.2 Classical Techniques

We review in this section classical techniques for failure analysis. That is, techniques that allow experts to model faults using knowledge of the system and domain knowledge about its basic failure behavior. These failure models can be created without a formal system model but must be re-evaluated every time a change to the system is done. In contrast, model-based approaches allow

the generation of classical fault models from a system model and some basic component fault models. Changing the system itself then requires a regeneration instead of a hands-on manual re-evaluation.

2.3.2.1 FMECA

Classical approaches include the Failure Modes and Effects Analysis (FMEA), which provides a structured technique to hierarchically decompose a system into its basic components and then examine them for failures and their causes in a bottom-up manner. This approach also refines the notion of fault to *failure modes*, which does not just entail that a fault occurred but also in which way it occurred. Typically, FMEA is also combined with a *criticality analysis*, creating the Failure Modes, Effects, and Criticality Analysis (FMECA). The FMECA also examines severity and probability of system failures [37] and computes a criticality value based on them. An example FMECA is shown in Tab. 2.2.

Item	Function	Failure Mode	Cause	Effect	Criticality
...					
Power	Energy Storage	battery charge too low	-	-	3 - major
Software	Surveillance	no housekeeping data	software task failed	no detection of anomalies	3 - major
...					

Table 2.2: Example entries of a simplified FMECA table. The criticality here is derived qualitatively from the failure mode.

Further information regarding the detection and the handling of failures, that is, the FDIR, is in many cases also directly embedded into the FMECA.

2.3.2.2 FTA

Another popular technique, employed in state-of-the-art failure analysis, is the FTA [1]. Since we also employ the fault tree analysis approach as our fault model of choice, we will review this particular manner of failure analysis more in-depth. In this technique, individual system failures are examined by considering them as the root of a tree and recursively examining combinations of faults that may lead to a higher level failure. In this manner, fault tree analysis models how faults of components propagate through the system and eventually cause a system failure. The approach itself is graphical and similar to modeling logical circuits, making it easy to visualize and intuitive to work with. Also, while they are qualitative models, they can be easily extended to include quantitative information. Most commonly, components are associated with failure probabilities to compute the

overall system failure probability. In many extensions, components are associated with a failure rate instead of a probability, which allows analyzing reliability metrics such as system reliability over time.

Syntactically, a fault tree is made up of *events* and *gates*. Events include the Top-Level Event (TLE), usually modeling system failure, intermediate events, and finally, the Basic Events (BEs), which form the leaves of a fault tree. Typically, BEs describe elementary faults on the component level that may occur at any time and intermediate events faults on a subsystem level. In the following, all non-basic events will also just be called faults. If a fault has multiple inputs, such as multiple basic events, propagation is interpreted with an OR logic. This ensures that a fault can have multiple reasons to occur, while also allowing to directly associate a basic event to the triggered fault. Gates then employ simple Boolean Logic to combine lower-level faults to higher-level ones. With this, statements such as “The system C fails if component A and component B fail” can be modeled and refined to arbitrary levels of precision.

Static Fault Trees (SFTs) The most basic case of a fault tree is known as a static fault tree, as it models no notions of timed dynamics or change and mainly employs the standard Boolean operators as gates.

SFTs use Boolean operations represented by AND and OR gates. Other gates exist, such as the k -VOTE gate, which propagates if at least k inputs have failed. Observe that a 1-VOTE gate corresponds to an OR gate and a k -VOTE gate with k inputs to an AND gate. Implementation-wise, all gates can therefore be considered as k -VOTE gates for some fitting k . Some other extensions also introduce a NOT gate. However, this allows the construction of fault trees where the TLE can go from having failed to operational again as new failures occur. These fault trees are known as non-coherent fault trees and have been dismissed as being a sign for modeling errors [38]. Fig. 2.7 shows the gates and events used in the SFT notation.

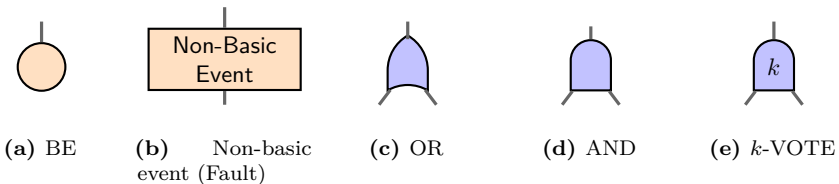


Fig. 2.7. Gates and events in a static fault tree.

2.3. TECHNIQUES FOR FAULT MODELING AND ANALYSIS

The reader may also be familiar with many more fault tree elements from the literature, such as the INHIBIT gate or the TRANSFER gates. Many syntactic propositions for improving fault tree modeling have been made, but we will restrict ourselves to the depicted gates and events for the examples used in this work.

To illustrate fault modeling with fault trees in general and static fault trees in particular, we consider the example fault tree depicted in Fig. 2.8. Here, we have a simple power system consisting of two hot redundant batteries and some cable connecting them to the main system. A battery can die if it is either no longer charged or damaged. Similarly, the power cable may rupture and lead to a loss of power. B1-B5 are the basic events that may fail at any time and lead to the intermediate faults described in the fault tree.

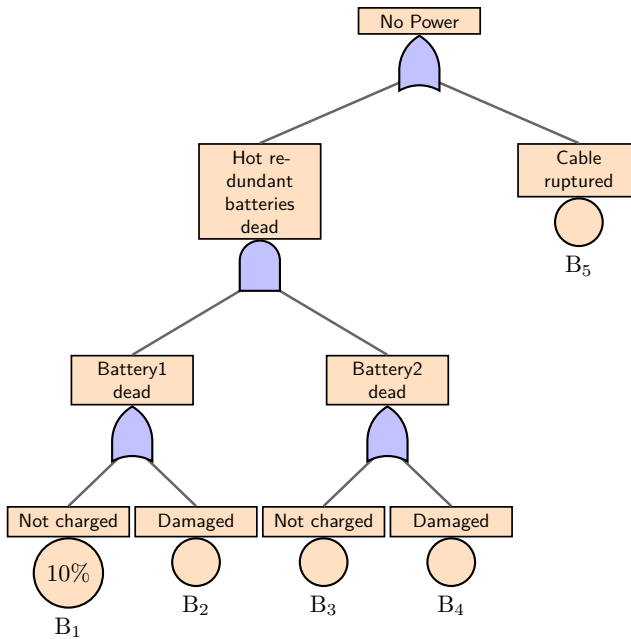


Fig. 2.8. Example of a (Static) Fault Tree. For B_1 a failure probability of 10% has been specified, for the other BEs the failure probability has been omitted.

From this example, it can also be seen how one can obtain quantitative information such as the system reliability from a given fault tree model. Given

the failure probabilities of the basic events B1-B5, one can calculate the likelihood of top-level event using standard combinatorial techniques. However, applying simple combinatorial approaches can perform poorly for larger fault trees as the number of fault combinations increases exponentially with the number of faults. Efficiently evaluating static fault trees can therefore require more sophisticated methods. The most common method is to transform them into Binary Decision Diagrams (BDDs) [39].

Dynamic Fault Trees While static fault trees allow for modeling simple Boolean relations between faults, they cannot handle faults whose behavior may change over time, depending on previously occurred faults. This, however, can be problematic and has led research to introduce the notion of time dynamic behavior into fault trees by extending the model of static fault trees. Dynamic fault trees can model temporal behavior, allowing for example to model that a set of faults is only malicious if they occur in a specific order. Furthermore, DFTs introduce the notion of spare management. In a static fault tree, only hot redundancy can be modeled, as all basic events can occur at all points in time. However, for cold or warm spares, this does not apply. As they are not in active use, or in the case of a cold spare, maybe even wholly dormant, their failure probability should lower while being in the dormant state. Furthermore, if a primary unit fails and has to activate a spare, then the failure rate should be employed as usual. DFTs enable modeling such temporal effects by extending static fault trees with additional temporal gates. Fig. 2.9 depicts the notation to extend SFTs to DFTs introducing new gates POR, PAND, SPARE, and FDEP.

The Priority AND (PAND) gate propagates if all inputs fail in sequence from left to right. It does not propagate the failure in case the sequence is not obeyed. The Priority OR (POR) gate propagates in case the leftmost input occurs before all other inputs [40]. Priority gates may usually come in two flavors: exclusive or inclusive. The inclusive PAND gate also propagates if the inputs occur simultaneously. On the other hand, the exclusive PAND gate only

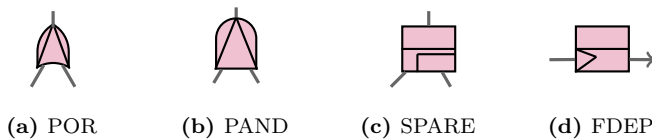


Fig. 2.9. Standard dynamic gates.

propagates if all inputs occur strictly after each other. Similarly, the exclusive POR gate only propagates if the leftmost input occurs strictly before all other inputs. It is shown in [41] that exclusive POR gates are expressive enough to model all priority gates. In this work, priority gates are considered to be exclusive.

The SPARE gate propagates a failure if the primary input failed and all spares are either claimed or failed themselves. The SPARE gate is connected to a primary event and a set of spare events. The spare events can be shared with another SPARE gate; therefore, a spare can be claimed by either the one or the other SPARE gate. However, there may be no shared elements between the primary input and any spare. We also allow for spares' nesting, e.g., SPARE gates can be spares.

The Functional Dependency (FDEP) gate has a trigger event on the left-hand side and any number of dependent events functionally dependent on the triggering event. When the trigger event occurs, the dependent events are also set to fail. The output of an FDEP gate only indicates to which tree it belongs and has no further semantic meaning.

A node is considered active iff there is either no SPARE gate on each path from the node to the TLE or if any SPARE gate on the path claims this node.

To avoid semantic problems, several additional syntactical restrictions to the fault tree structure are imposed. We say that a fault tree is well-formed iff:

- It has exactly one root element, the top-level event.
- Spares may be shared but their subtrees may not have common child nodes with other subtrees. FDEPs, however, may have dependent events across different spare subtrees.
- FDEPs may have any event as a trigger event but may not induce loops through dependent events.

The Priority AND (PAND) gate is similar to the AND gate but dictates that the input fault events must occur in the order from left to right. The SPARE gate allows the modeling of spares as described above. The first input is always the primary unit and active per default, that is, all faults in the subtree of the primary fault input can occur as before. The other fault inputs are considered spares and may be dormant. In DFTs, basic events are usually no longer assigned just a failure rate but also a dormancy value ranging between 0 and 1. Multiplying dormancy and failure rate then gives the reduced failure rate for an inactive spare. For DFTs, many other gates have been introduced, such as

the SEQ gate, which enforces fault events to occur in a specific order. Also, other types of spare gates such as hot spare gates, warm spare gates, and cold spare gates have been considered, allowing to determine the dormancy in dependency of the overlying spare gate. For further reading on the rich extensions DFTs provide and also the semantic problems these entail for evaluating them, we refer the reader to [41].

In the following, only well-formed fault trees are considered. To illustrate the DFT notation, an example DFT will be considered now. Fig. 2.10a shows a use case for the PAND gate. The system itself is depicted in Fig. 2.10b and consists of some primary equipment, a cold spare unit, and a switch that switches to the cold spare should the primary unit fail. According to DFT semantics, the system is non-failing if the switch fails after the primary component. However, should the switch fail before the primary, then we are unable to switch to the redundancy.

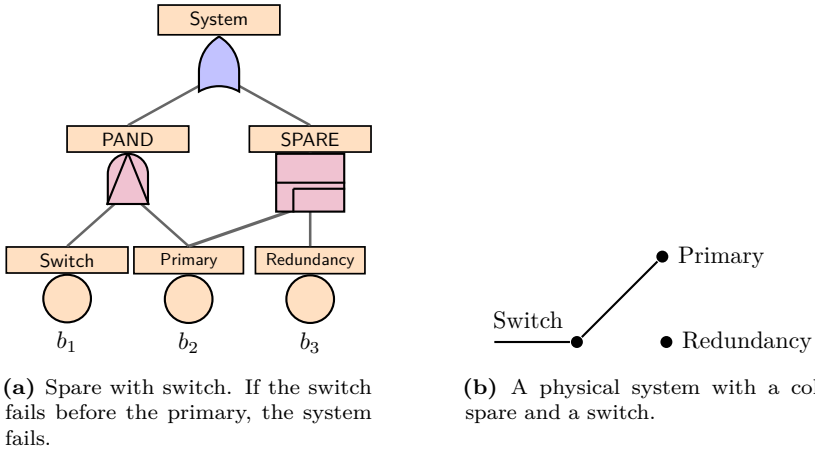


Fig. 2.10. (a) shows an example DFT with a PAND gate; (b) is the system represented by (a)

As with SFTs, it is also possible to quantitatively evaluate DFTs, but this requires more involved techniques. Standard approaches include the transformation of a DFT into a Markov chain and then computing the probability of reaching a state where the top-level event occurs [42]. We will also see later that Markov chains are insufficient models when we consider DFTs with shared spares and spare races become a possibility.

Repairable Fault Trees We finally consider one more interesting extension to fault trees, namely the notion of repairability. In standard fault trees, fault events may only occur, but there is no notion of repairing faults. This makes it challenging to model recovery mechanisms in a fault tree. While SPARE gates in a DFT allow very simple recovery by replacement, recovery utilizing repair is outside the scope of classical fault trees. Repairable Fault Trees (RFTs) aim to fill this gap. If the on-board computer receives too much faulty data for the voter to mask, the sensors providing the data will be repaired (for example, by restarting the sensor). For a survey giving further insight into the state-of-the-art techniques and model extensions for fault trees, we refer the interested reader to [2]. Repair also has consequences on the semantics of the priority gates, as it is not clear what the semantical behavior is when a child of a priority gate is repaired.

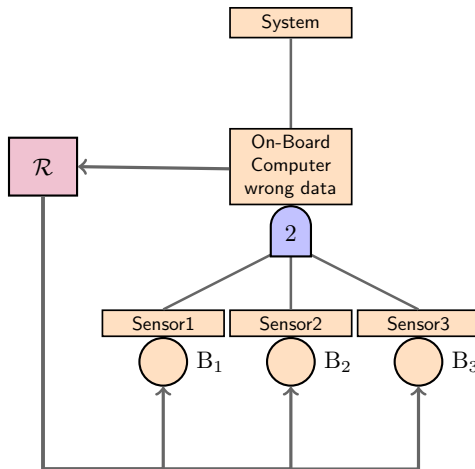


Fig. 2.11. Example of a Repairable Fault Tree using a Repair Box \mathcal{R} . If more than one sensor fails, \mathcal{R} will decide based on some internal logic which sensor needs to be prioritized.

2.3.2.3 Reliability Block Diagram Method

The Reliability Block Diagram (RBD) method [43] functions very similar to fault tree analysis, but instead of forming a graph modeling the fault propagation in a tree-like fashion, RBDs model paths of errors in a fashion based on resistors

in electrical circuits. Each block in an RBD models a fault-tolerant unit that must fail for the entire system to fail. The blocks can be arranged serially, thus enabling RBDs to model temporal behavior. In a parallel configuration, all parallel paths have to fail for the entire parallel composition to fail, enabling the modeling of hot spares.

Fig. 2.12 shows an example RBD modeling a system with a primary unit that switches to a cold spare if it fails. Should the first cold spare fail, it switches to a system of two hot spares. Should both of these fail as well, the system uses another cold spare unit. Should this unit fail as well, then the system fails.

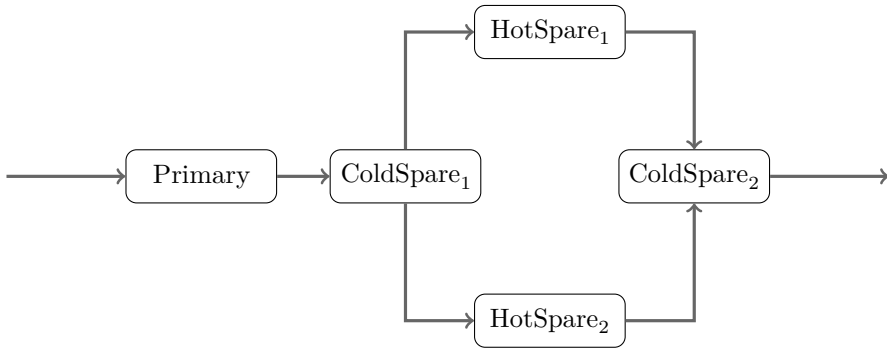


Fig. 2.12. Example Reliability Block Diagram. In order for the system to fail, the primary must fail, then ColdSpare₁, then either HotSpare₁ or HotSpare₂ and finally ColdSpare₂.

An extension of RBDs to Dynamic RBDs (DRBDs) is given in [44]. DRBDs function in a similar way to DFTs by also providing features such as spare management with shared pool. Using a switching mechanism, the spares can be switched from one RBD into another RBD.

2.3.2.4 Combined Techniques

Combinations of these techniques such as FTA and FMECA to form higher-level analysis methods closing the gap between bottom-up and top-to-bottom analysis, such as the Probability Risk Assessment (PRA) have also been considered and accepted by the industry [45]. Also, a more component-oriented approach for combining FTA and FMECA, the Fault Propagation and Transformation Notation (FPTN), has been proposed in [46].

2.3.2.5 FPG

Modern research has put significant effort into improving these basic methods to incorporate additional phenomena relevant to the failure behavior. In particular, a major concern is time. In a system that may eventually fail, it is not only of interest how it may fail but also after what time span. Techniques aiming towards providing an answer to this issue have also been developed. The basis for many such state-of-the-art approaches are Fault Propagation Graphs (FPGs), which attempt to provide a graph theoretic approach to modeling fault behavior by focusing on the description of the fault propagation. In this setting, the nodes are failure modes (sometimes also extended by other system modes, basic fault events, and more), and edges between nodes represent the fault propagation from the source to the destination node [47].

An extension of this formalism to also include timing aspects, called Timed FPG (TFPG), has been presented in [48] and has been successfully applied in the European Space Agency (ESA) project FAME, not only to analyze failures but also to synthesize appropriate FDIR components to combat them [49]. Fig. 2.13 shows an example of a TFPG. Circled nodes are basic fault events, while others are failure modes. Edges are labeled with the minimum and maximum propagation time.

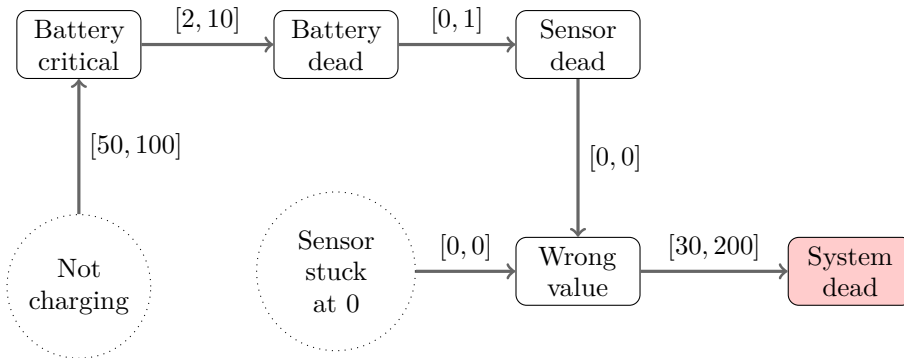


Fig. 2.13. Example of a simple TFPG. After 50 to 100 time units the Not charging fault will occur and propagate through the graph.

2.3.2.6 HAZOP

These approaches share the common trait of requiring domain knowledge about basic fault events. However, how does the reliability engineer identify a basic fault event and that he has covered the relevant ones? An industry-accepted technique for this is the Hazard and Operability Studies (HAZOP), for which an overview can be found in [50]. HAZOP is a means of imaginative anticipation of hazards. A standard set of *guide words* is defined based on past experiences, prompting analysts to consider how these guide words might manifest themselves on a basic level. Originally hailing from the field of chemistry, HAZOP has been implemented in many other safety-critical fields such as the development of nuclear power plants and has also been adapted to be employed in hardware and software systems [51], where it is called Software Hazard Analysis and Resolution in Design (SHARD). The following is a possible example of a set of guide words and is employed by FPTN:

- Omission: A message was expected but not delivered.
- Commission: A message that should not have been delivered has been delivered.
- Value: An incorrect value has been computed/received/sent.
- Timing: A message is too early, too late.

To obtain more detailed analysis results, the guide words may be further refined (e.g., value subtle incorrect and value coarse incorrect).

These manual approaches, however, are labor-intensive and often not incremental. In fact, slight changes to the system model may invalidate them entirely, requiring them to be redone from the ground up. As such, they are unsuitable to be used in the early stages of spacecraft design, where many components are still subject to change. Many modern approaches have therefore adopted a model-based direction so that not only the system is represented with a formal model but also its erroneous behavior, thus allowing for failure analysis and validation of system models to be performed automatically.

2.3.3 Model-Based Techniques

Early pioneers of this approach come from the field of artificial intelligence, who developed the framework of Model-Based Diagnosis (MDB) [52] to create FDIR systems capable of performing on-line diagnosis using logical inference methods.

In MBD, a faulty state is represented by an "abnormal" predicate encoded by an equation system of analytical redundancies. The fault behavior is then analyzed on a component level and given to the MBD engine via logical formula. System-wide failure behavior can then be automatically diagnosed (or at least attempted) by employing the MBD as the Decision Logic in a model-based FDIR system.

This approach of modeling faulty behavior on a component-wide level and leveraging the knowledge to compute the failure behavior on the system level has also continued into modern failure analysis methods.

Of particular note is the FPTN based Fault Propagation and Transformation Calculus (FPTC) introduced in [53]. Here, an architectural system model is taken, and each component is annotated with its local failure behavior expressed by local transformation rules. These rules map combinations of incoming faults together with internal faults to fault propagations into other components. The basic workflow is to consider each component in isolation and then use a set of guide words as in HAZOP to create the local transformation rules. As usual, the system-wide failure behavior can then be automatically calculated. The following demonstrates a simple set of transformation rules for a sensor similar to the one from the TFPG example.

$$\begin{aligned} * &\rightarrow \text{output.value.stuckAt0} \\ \text{power.omission} &\rightarrow \text{output.omission} \end{aligned}$$

The wildcard $*$ indicates that a fault is generated from this component and may model a component facing an internal failure. FPTC itself is originally a qualitative method, but quantitative extensions have also been proposed. For example, in [54] a probabilistic extension to the FPTC framework is proposed, enabling the analysis also to determine system-wide failure probabilities.

However, while these methods all enjoy the benefit of being employable while the system design is still in its early phase, its heavy focus on explicitly modeling the failure processes while treating the components as black boxes makes them heavily reliant on the domain experts operating them and providing their domain knowledge of the failure behavior. More comprehensive approaches employing state-of-the-art model checking techniques to determine precise system-wide error behavior from a (possibly) complex system model while relying only on a simple error model have also been developed. For example, in the COMPASS toolset, very detailed system models can be considered, taking into account not only the basic system architecture but also the components implementation in software and hardware [55]. The drawback of this approach is that its feasibility requires

a mature system model, which may only be available at a later design stage. The FAME toolset building onto COMPASS attempts to overcome this limitation by starting with black-box components for its initial analysis and generating a COMPASS model when the design has reached the necessary maturity status.

2.4 From Fault Model to Recovery Strategy

Computing strategies for recovery purposes from a given fault model has been researched in other contexts. The original authors of the DFT model recognized the issue of spare races and extended their original proposition [42] by randomly resolving the race using a uniform distribution.

In [62], the authors encode the non-determinism into the failure propagation of FDEP gates. By resolving the FDEP gate triggers sequentially, spare gates do not truly fail simultaneously and can thus claim according to their deterministic semantics. The non-determinism is then resolved by computing the worst-case order of failure propagation. The core difference between the approach proposed by the authors and the approach proposed in this work lies in the non-determinism here being applied to the spare gate claiming.

An approach similar to ours is taken in [56], which focuses on repairable fault trees. The authors consider non-deterministic repair policies where the order of repair operations is not fixed. Repairable fault trees are an extension to fault trees where faults can be transient or repaired. In their work, the repair process is realized with a new gate type called *repair boxes*. Repair boxes are equipped with a repair policy that states which resources are required for the repair process, in which order the faults should be repaired, the repair rate, and so on. By then converting a repairable fault tree to a Markov decision process, an optimal repair policy (with respect to the steady state availability metric) can be computed. However, the authors do not consider DFT models.

Also focusing on repair policies and in addition on maintenance policies, [57] introduces a model of fault maintenance trees, aiming to identify optimal repair and maintenance strategies. The model supports inspection boxes, which can be equipped with an inspection policy, such as checking the observation state after certain time intervals. Repair policies can be modeled free-form using Input-Output Interactive Markov Chains (I/O-IMCs). The semantics are defined compositionally by starting with elementary I/O-IMCs for every gate and proceeding in a bottom-up direction by combining these elementary I/O-IMCs. Recovery strategies for resolving spare races can also be given on the

Markovian level. The strategies must be chosen manually and then compared using a testing-based approach based on model simulation.

Dynamic Decision Networks (DDN) are employed in [58], and their inference capabilities are exploited to create autonomous on-board FDIR systems for spacecraft that can select reactive and preventive recovery actions during runtime. The authors further consider in [59] how DDNs can be generated from an extended DFT model. Instead of off-line computing a recovery strategy with globally optimal reliability, the approach focuses on providing locally optimal (in terms of some externally provided heuristic utility functions) on-line decision making.

Building upon the COMPASS [60] toolset, the FAME [49] process uses Timed Failure Propagation Graphs to synthesize FDIR components. These components are monitors for fault detection and fault recovery units that implement recovery plans for each specified combination of fault and spacecraft mode. Here, planning-based approaches with predefined actions are employed to create the recovery components. For the detection synthesis, the FAME process provides an algorithm that generates a set of alarms necessary to distinguish different faulty states. These alarms are then fed into the synthesized recovery plans. The generated components are then transformed into a COMPASS compatible representation in the form of SLIM [60] models. A developed understanding of the system implementation is needed to provide the required timing information. Many FDIR concepts such as redundancy features, however, are ideally already finalized at this stage.

For aircraft systems, the authors of [61] model loss-of-control scenarios using MDPs. They synthesize optimal strategies but report on struggles to represent the resulting policies due to large state spaces.

The problem of failure rates sometimes not being known a priori is tackled in [62]. The authors consider fault trees with symbolic failure rates and synthesize upper failure rate bounds for meeting reliability thresholds.

2.5 Partial Observability

Most previously considered models and techniques share the common trait of focusing on fully observable models. Models where it is not always known that an event, such as a fault, has occurred and first needs to be observed to gain that knowledge, are called *partially observable*. Likewise, this also gives us the information on which fault is responsible for causing a failure and thus helps in

deciding which recovery action should be taken to return into an operational state, or if that is not possible.

Bayesian-based models natively support such inference and can be employed to deduct possible fault causes as shown in the DDN approach from [58]. Partially Observable Markov Decision Processes (POMDPs) [63] extend MDPs by introducing partially observable states by allowing each state to be assigned a set of observations. Based on this, there exists also the notion of a Belief MDP where each state models the belief of being in a certain state of a POMDP. Usually, this is achieved by equipping a belief state with a mapping assigning each POMDP state a probability value (the belief). However, computing properties such as total long-term reachability necessary to derive interesting metrics such as MTTF is known to be undecidable [64]. Intuitively, this stems from the belief MDP being potentially infinite and optimal policies requiring infinite histories to decide the optimal action.

The idea of a continuous-time extension of a POMDP to a Partially Observable Markov Automaton (POMA) has to the best of our knowledge not been successfully formalized.

On the level of fault trees, the authors of in [56] propose the idea of a partially observable, non-deterministic repairable fault tree on the basis of POMDPs in their future work section. To the best of our knowledge, however, this idea has not received a follow-up in subsequent papers.

The authors of [57] propose the introduction of an Inspection Module (IM), which marks an event as observed after a certain time period and only permits actions to be taken on observed events. Beliefs are not taken into account, therefore, the decision process does not change when a fault is very likely to have occurred, even if it has not been observed.

Overall, the literature suggests that while certain aspects of partial observability have been handled, there is in particular a gap in defining semantics for claim actions on SPARE gates under uncertainty.

Chapter 3

Preliminaries

In this chapter, we introduce the formal notations and definitions relevant to the thesis. We focus mostly on the mathematical nomenclature required for proofs and the formal definitions for Markov structures and Fault Trees.

3.1 Basic Notation

In most cases, single elements are denoted with lowercase and sets with a capital letter. The set operations \cup (union of sets), \cap (intersection of sets), \setminus (set exclusion), and \times (cross product of sets) are defined as usual. The special union symbol \uplus denotes the union of disjoint sets. The power set 2^A denotes the set containing all subsets of A . The empty set is denoted as usual by \emptyset .

\mathbb{N} and \mathbb{R} denote the natural and the real numbers, respectively. With $\mathbb{R}_{\geq 0}$ we denote the set of non-negative real numbers and likewise with $\mathbb{R}_{> 0}$ the set of positive real numbers without 0. An interval $[x, y]$ defines the usual subset of numbers over \mathbb{R} . A mapping f between sets A, B is written as usual as $f: A \rightarrow B$. A is then called the domain and B the codomain. To access the codomain of a mapping f , we employ the usual notation $Codomain(f) := B$. We also sometimes use the notation $f: a \mapsto b$ to write mappings where we omit the domains. A tuple $t = (a, b)$ denotes as usual an element from $A \times B$. For such a tuple we sometimes also write $a(t) := a$ or $b(t) := b$ to access the tuple elements. For mappings defined on tuples such as (a, b) we simply write $f((a, b)) = f(a, b)$.

The Boolean operations \wedge (logical and), \vee (logical or), and \neg (logical negation) are also defined as usual. The quantor $\exists a \in A: \varphi$ denotes as usual that an element $a \in A$ must exist to satisfy φ . Likewise, the quantor $\forall a \in A: \varphi$ denotes that φ must be satisfied for any $a \in A$.

A total, or linear, order denotes an ordering that relates all elements in a set by a $<$ relation. Formally:

Definition 3.1 (Total Order). *A total order is a tuple (A, \leq) with $\leq \subseteq A \times A$ such that for any $a, b, c \in A$:*

- $a \leq a$ (Reflexivity),
- $a \leq b$ and $b \leq a$ then $a = b$ (Antisymmetry),
- if $a \leq b$ and $b \leq c$ then $a \leq c$ (Transitive), and
- if $a \neq b$ then $a \leq b$ or $b \leq a$ (Totality).

Sometimes elements from a set are also called words. In this case, the set is also referred to as a language. The concatenation of words u, v is denoted as usual by $u \cdot v = uv$. Similarly, the concatenation of all words between two languages A, B is denoted as AB . The set containing all finite word concatenations over a language A is denoted as Kleene Closure $A^* := \bigcup_{i \in \mathbb{N}} A^i$ with $A^0 = \{\epsilon\}$. The special character ϵ is also called the empty word with $u \cdot \epsilon = u$ for any word u .

A (probability) distribution d over a set A is a mapping $d: A \rightarrow \mathbb{R}_{\geq 0}$ such that $\sum_{a \in A} d(a) = 1$. We denote the set of all probability distributions over A with $Dist(A)$.

3.2 Markovian Structures

In this work, we will only require CTMCs and Markov Automata. Further Markovian structures such as DTMCs or MDPs are omitted here.

3.2.1 Markov Chains

We mostly rely on the common definition for Markov Chains, extended with an initial distribution and a labeling function:

Definition 3.2. *A (Continuous Time) Markov Chain (MC) is a 5-tuple (S, T, I, L, R) where:*

- S is a set of states.
- $T \subseteq S \times S$ is a set of transitions.
- I is the initial probability distribution over S .
- $L: S \rightarrow 2^D$ is a labeling function over S over a domain of labels D .
- $R: T \rightarrow \mathbb{R}_{>0}$ is the transition rate function.

The exit rate of a state s of a Markov Chain \mathcal{C} is denoted as $\text{exit}_{\mathcal{C}}(s) = \sum_{(s,s') \in T} R(s,s')$. If the context is clear, we drop the subscript \mathcal{C} . As we only employ CTMCs, we will abbreviate them simply as MCs.

3.2.2 Markov Automata

The formal definition of a Markov Automaton is based on [33]. In the original definition, non-deterministic and probabilistic transitions are captured in one relation. For the later constructions it will be convenient to distinguish between states with only non-deterministic transitions, Markovian states, and probabilistic states. We therefore split the relation for the purpose of this work into two. In addition, we also add a labeling function in the style of MCs. Overall, we capture it by the following definition:

Definition 3.3. *A Markov Automaton (MA) is a 7-tuple (S, L, A, N, C, P, s_0) where:*

- S is a set of states.
- $L: S \rightarrow 2^D$ is a labeling function over S over a domain of labels D .
- A is a set of actions.
- $N \subseteq S \times A \times S$ is a set of non-deterministic immediate transitions.
- $C \subseteq S \times \mathbb{R}_{>0} \times S$ is a set of exponentially delayed transitions. We restrict C such that if $(s, \lambda, s') \in C$, then $(s, \lambda', s') \notin C$ for any λ' . (There is at most one exponentially delayed transition to the same state.)
- $P \subseteq S \times \text{Dist}(S)$ is a set of probabilistic immediate transitions.
- $s_0 \in S$ is the initial state.

If a state has non-deterministic transitions, we call it a *non-deterministic state*, a state with exponential delayed transitions is a *continuous state*, and a state with probabilistic transitions is a *probabilistic state*. Hybrid states that combine different transition types are not allowed. Note that for a Markov Automaton, a fixed initial state s_0 is sufficient as an initial distribution can be modeled by adding corresponding probabilistic transitions from s_0 .

3.3 Dynamic Fault Trees

Since the syntactic difference between Dynamic Fault Trees and classic Static Fault Trees lies in the addition of new gates, we first formalize a general notion of a fault tree that does not depend on the used gate types. Following the definitions of [2, 56], a fault tree is formalized as follows:

Definition 3.4. (*Fault Tree*) A Fault Tree (FT) $\mathcal{T} = (F, B, G, T, P, D)$ is a structure with the following constituents:

- F is a set of faults.
- B is a set of basic events.
- G is a set of gates.
- $T : G \rightarrow \mathbf{GateTypes}$ is a mapping assigning each gate a type.
- $P \subseteq (F \cup B \cup G) \times (F \cup G)$ describes the propagations within the fault tree.
- $D : B \rightarrow (\mathbb{R}_{>0} \times [0, 1] \times \mathbf{EXP}) \cup ([0, 1] \times \mathbf{DISCRETE})$ is a function assigning a basic event its failure distribution. This can be either
 - continuous-time exponential $(\lambda, d, \mathbf{EXP})$ with rate λ and dormancy d , or a
 - discrete probabilistic $(p, \mathbf{DISCRETE})$ with probability p .

Furthermore, the induced graph $(F \cup B \cup G, P)$ must

- be acyclic,
- follow the individual syntactical restrictions for each given gate type,
- and must have a unique root, the TLE $e \in F$.

Generally, basic events will be denoted by b_1, b_2, \dots , sets of basic events by B_1, B_2, \dots and failure rates by $\lambda_1, \lambda_2, \dots$. As for the association of failure rates with basic events, in the following, for any basic event b with an exponential distribution, the active failure rate will be denoted by $F_A(b)$ and the dormant failure rate by $F_D(b) := F_A(b) \cdot d$ where d is the dormancy of b . In the case of $F_A(b) = F_D(b)$, the subscripts will be dropped, and the simplified notation $F(b)$ will be used to denote the failure rate.

When given an FT \mathcal{T} we also write $F(\mathcal{T}), B(\mathcal{T}), G(\mathcal{T}), T(\mathcal{T}), P(\mathcal{T}), D(\mathcal{T})$ to identify the associated elements in the tuple. To identify the unique root of an FT, we also write $\text{TLE}(\mathcal{T})$. The set of all nodes in the tree is denoted by $N(\mathcal{T}) := F \cup B \cup G$.

For a DFT, we consider the gate types introduced in Section 2.3.2.2. This gives us the following set:

$$\text{DFTGates} = \{\text{OR}, \text{AND}\} \cup \{k\text{-VOTE} \mid k \geq 1\} \cup \{\text{POR}, \text{PAND}, \text{SPARE}, \text{FDEP}\}$$

For the priority gates POR and PAND, we pick a repair semantics where repair events act as a general inverse to a fault event, that is, the occurrence of a repair event puts the gate into a state as if the original failure event had never occurred. Or in other words: Priority gates do not have to be repaired in order. We pick this semantics as we will later track the occurrence of events in a history variable and allowing said history variable to remember all repair events, would lead to an infinite state space. By employing the set DFTGates as the gate types in the FT definition, we now obtain the formal definition of a standard DFT:

Definition 3.5. (*Dynamic Fault Tree*) A (standard) Dynamic Fault Tree (DFT) \mathcal{T} is an FT with $\text{GateTypes} = \text{DFTGates}$.

Since the semantics of DFTs are a subject of research of this thesis, different perspectives for defining them are discussed in the main Chapter 4.

Chapter 4

Formalization of the FDIR Model

In this chapter, we discuss important features the FDIR model needs to possess in order to deal with FDIR scenarios for space systems, and what weaknesses DFTs have to capture them. We then formalize an extension to the DFT model to deal with these weaknesses.

While systems are growing in complexity, they are at the same time decreasing in size. There is an increasing demand for low-cost space missions, mostly carried out using the popular small satellite CubeSat architecture [65]. These CubeSats, however, face enormous challenges regarding reliability. The authors of [65] report that the average mission success chance of a CubeSat mission as in 2018 was 65%, as opposed to a mission success chance of 85% for large-scale satellite missions. The low-cost requirements and lack of space make it difficult to deal with this deficit in reliability using traditional engineering techniques such as redundancies. This gives rise to a need for more complex FDIR strategies that better exploit existing resources.

Examples of such advanced FDIR strategies are sharing redundancies, reusing measurement outputs of sensor units for constructing analytical redundancies and integrating repair mechanisms. Accompanying the need for such complex FDIR strategies also comes the need to verify and validate their correctness before mission start. Both to potentially find design problems and to raise trust in the FDIR system despite its increase in complexity. A suitable FDIR model for space missions would need to be able to deal at minimum with the following scenarios:

- Redundancies, including sharing of redundancies
- Repair mechanisms (e.g., patching software bugs)
- Transient faults (e.g., transient bitflips)
- Limited information, dealing with only partial information

As shown in Section 2.3, there exists a wide range of techniques for modeling and analyzing FDIR. With their ability to naturally model redundancies, existing extensions for repair, and their intrinsic support for Boolean and Temporal connections, DFTs give us a suitable range of features needed to cover the needs of modeling FDIR for spacecraft.

However, when it comes to modeling more complex relations between components, default fault tree semantics quickly become too rigid. DFTs impose a fixed and rigid order in which spares are activated. They do not allow to adapt the order depending on the history of occurred faults. This may lead to semantically undesirable consequences:

- A SPARE gate might claim a spare from a spare pool, despite having an already failed parent. This might deny a necessary resource to other SPARE gates that urgently require the spare to recover.
- In the event of spare races, it is not semantically clear which SPARE gate may claim a spare.
- The optimal order for spares has to be known at design time.

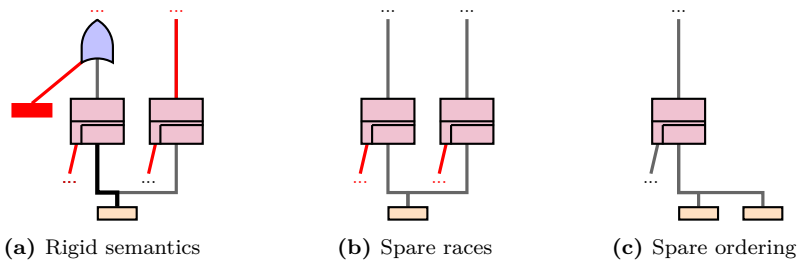


Fig. 4.1. Example configurations of problematic DFTs. Red indicates an incoming failure propagation. Spare claims are marked with thick, black lines.

Fig. 4.1 on the previous page visualizes possible fragments of DFT configurations exhibiting the described semantic complications. A simple, but full example where different resolutions of spare races lead to different behavior with different MTTFs can be found later in Fig. 4.6 on page 46

The problem cases induced by the rigid standard fault tree semantics have been considered in other works. The authors of [41] tackle the issue of spare races by employing non-determinism in the propagation semantics of functional dependency gates while allowing only functional dependencies to cause spare races. The study of the interaction of this approach with spare gates reveals various yet sensible ways in which the resulting semantics can be interpreted. They conclude that there is no “correct” one-fits-all interpretation and that the fitting variant has to be chosen on a case-by-case basis. This is a concern regarding the applicability of fault trees, as experts in system design are not necessarily experts in fault tree semantics.

This conflict between the applicability of different fault tree semantics leads to the following proposition: Instead of indirectly encoding the recovery behavior via the semantics into the fault tree model, we explicitly describe the recovery behavior within a special recovery model.

However, this now implies that the engineering expert needs to model not only the fault tree but also the recovery model. This in return demands knowledge about which actions are indeed the most suitable ones. This gives us the core question of this work:

Question: Can we obtain an “optimal” recovery strategy from a non-deterministic DFT?

The term optimal has been put into quotation marks here since it warrants some discussions to define what it means for a recovery strategy to be optimal. The main goal is to increase metrics used to measure system reliability and availability, but we will see in the following that the different metrics may cause undesirable behavior or make the approach non-tractable.

In order to systematically tackle this question, this chapter focuses on establishing a suitable formalized framework that allows deriving the optimal Recovery Automaton using model checking techniques. To simplify the definition process, the following chapters first operate under assumption of full observability. An extension of the formal concepts, including partial observability, is then constructed from this basis.

The overall strategy is to define an inherently non-deterministic DFT semantics using Markov Automata, which then yields optimal schedulers that in return

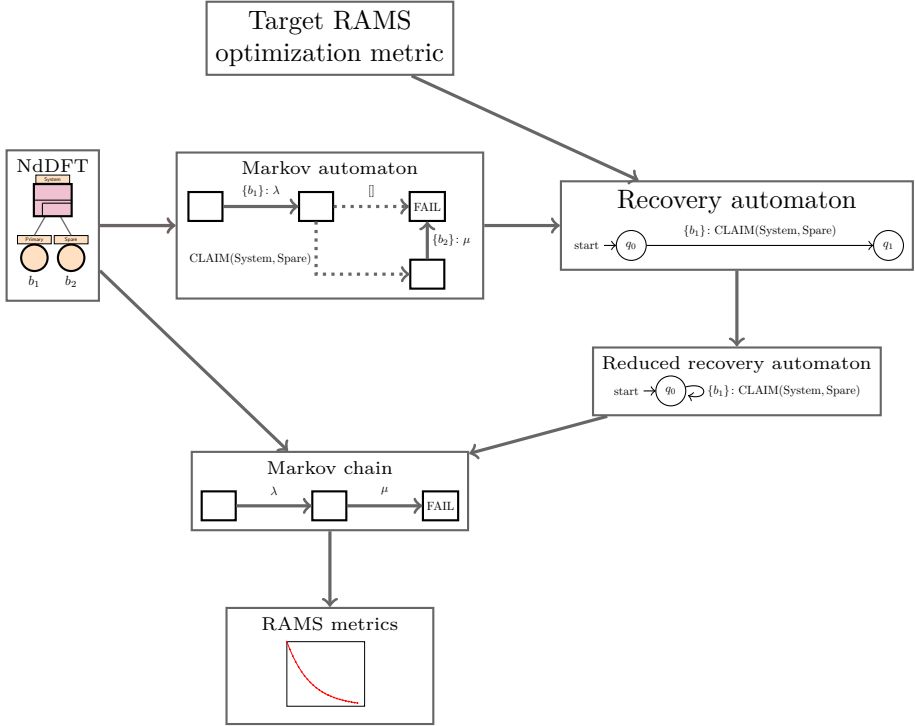


Fig. 4.2. Transformation road map. From the NdDFT model, a deterministic recovery model is derived. The composition of the recovery and the NdDFT model then gives a traditional Markov chain. Note that we have one fixed target RAMS metric, but may also want to evaluate the resulting Markov chain with other RAMS metrics different from the optimization target.

can be transformed into the desired Recovery Automaton. A road map that sketches the objects and transformations involved in the entire process is given in Fig. 4.2.

To build the formal basis of this transformation road map, we define the necessary semantics of the objects and the involved composition semantics in the following. E.g., we define what it formally means to use a Recovery Automaton conjunction with a DFT. Further processing of these objects and putting them together to achieve the desired framework is then done in the following chapter.

4.1 Rate Dependency Extension

The standard gates used in DFTs lack convenient tools for modeling dynamic failure rates. Due to various effects, such as degradation over time, failure rates are not always constant. We introduce the Rate Dependency (x -RDEP) gate depicted in Fig. 4.3 to provide some basic handling for modeling event-based failure rate increases.

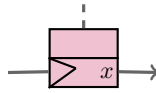


Fig. 4.3. Rate Dependency Gate. When a left-hand side input occurs, then the failure rate of all right-hand side basic events is multiplied by x .

The x -RDEP, based on [57], is structured similarly to the FDEP gate. When the triggering event on the left-hand side occurs, the dependent events have their failure rates multiplied by the rate dependency $x \in \mathbb{R}_{\geq 0}$. The RDEP gate also has a dummy output indicating to which tree it belongs. Like the FDEP gate, the output gate has no further semantic meaning. Using the RDEP gate, degeneration of components can be modeled. In reality, components do not have constant failure rates. Instead, the failure rate might increase with time due to wear and damaging influences. An RDEP can be a simple means to model such an increase. A second use case for the RDEP gate is modeling load-sharing of components. For example, if a system is powered by two batteries and a battery fails, then the system may continue operation, but the increased load on the remaining battery might increase its likelihood to fail. The two use cases are illustrated in the examples of Fig. 4.4 on the next page.

In the first example, an equipment has its failure rate doubled when the *Wear* event fires. In the second example, two hot redundant batteries are used to power a system. If one of them fails, the system is still operational, but due to the increased load, the failure rate of the remaining battery is doubled.

To avoid semantic issues of giving immediate events an occurrence probability of > 1 , RDEPs may only trigger basic events with exponential distributions. Updating the common DFT gate type definition with the family of RDEPs yields the following set:

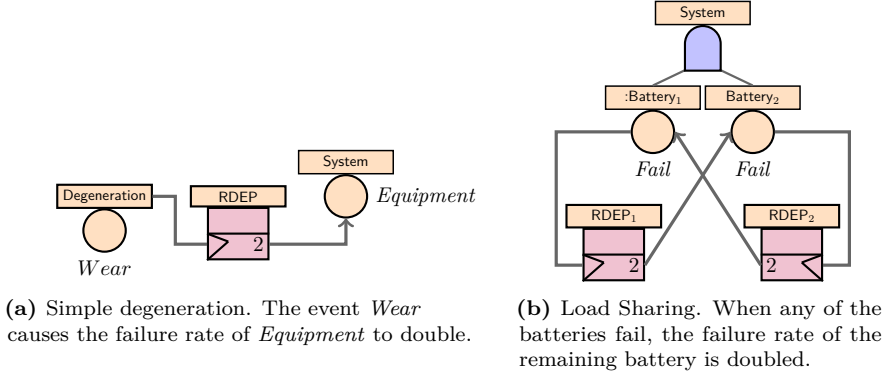


Fig. 4.4. Example of two use cases of RDEPs.

$$\begin{aligned}
 \text{DFTGates} := & \{\text{OR}, \text{AND}\} \cup \{k\text{-VOTE} \mid k \geq 1\} \\
 & \cup \{\text{POR}, \text{PAND}, \text{SPARE}, \text{FDEP}\} \\
 & \cup \{x\text{-RDEP} \mid x \in \mathbb{R}, x > 0\}
 \end{aligned}$$

4.2 Non-Deterministic Fault Trees

As described in the previous sections, DFTs require that spares are activated in a fixed and rigid order. This order cannot be adapted depending on faults that have previously occurred. Additionally, in cases of spare races, it is not semantically clear which SPARE gate claims the actual redundancy. To relax on this semantic restriction of the DFT model, an inherently Non-deterministic Dynamic Fault Tree (NdDFT), following the naming in [56], needs to be defined. This definition introduces a recovery strategy that can be optimized by first transforming the NdDFT into an MA. Computing an optimal scheduler for this MA using standard algorithms allows deriving a so-called Recovery Automaton (RA). This recovery automaton provides the optimal strategy to react to failures in the NdDFT. The interaction between the NdDFT and the recovery automaton is depicted in Fig. 4.5 on the following page.

On the one hand, the NdDFT models the non-deterministic fault behavior of the system. On the other hand, the RA corresponds to an abstraction of the

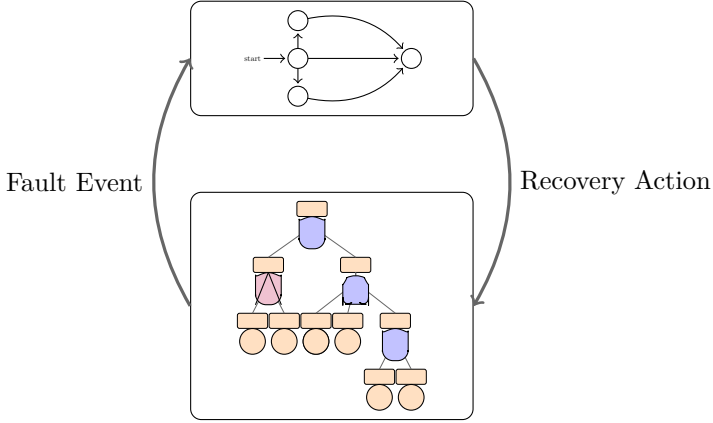


Fig. 4.5. Abstraction of System-FDIR interaction by FT-RA interaction.

FDIR logic. Interaction between the two is carried out via listening to events and firing recovery actions.

The NdDFT defined here is based on the same semantics as a DFT, except for the activation conditions of spares. The NdDFT drops the requirement that spares are always activated from left to right. The new non-deterministic semantics allow for a SPARE gate to not claim any of the attached spares, thus leaving it available for more critical SPARE gates that may also require a spare. The syntax and notation of the NdDFT is wholly adopted from the DFT. Whenever a fault event, or more precisely a BE, occurs in an NdDFT, the new semantics allow performing any valid recovery action of the following form:

Definition 4.1 (Recovery Action). *A recovery action r in an NdDFT \mathcal{T} is an action of the form*

- $[]$ (empty action),
- $CLAIM(G, S)$ (SPARE gate G claims spare S) with $(S, G) \in P(\mathcal{T})$, or
- $FREE(S)$ (all claims on spare S are removed) with $S \in N(\mathcal{T})$.

The recovery automaton will trigger these recovery actions upon receiving a set of basic events as input. Here, sets of basic events rather than single basic events are considered since FDEPs may cause several basic events to fail

simultaneously. As a helpful notation, define the set of all non-empty basic event sets as:

$$BES(\mathcal{T}) = \{B \subseteq B(\mathcal{T}) \mid B \neq \emptyset\}$$

Likewise, define the set of all recovery actions possible in an NdDFT by $R(\mathcal{T})$. To define the semantics of an NdDFT in terms of Markov Automata, we first need to formalize the necessary components making up a state of the NdDFT. In order to uniquely identify a state, we memorize the following information:

- A history of the occurred basic event sets, and
- a spare to SPARE gate mapping to memorize claims.

Note that it would also be possible to memorize the fail state of each node rather than the event history. However, since dynamic gates are allowed to consider the occurrence order of events to decide whether they propagate or not, this information may not be sufficient. For the introduced POR and PAND gates, it would be possible to discard the history as it is only necessary to memorize whether the occurred events are still in order. In order to free the formalization from these implementation details, we use the more powerful history information. Formalizing the above gives us the definition of a DFT state:

Definition 4.2 (DFT State). *A DFT state is a tuple $s = (\text{history}, \text{claims})$ with:*

- $\text{history} \in BES(\mathcal{T})^*$
- $\text{claims} \subseteq G(\mathcal{T}) \times N(\mathcal{T})$ a set of claims.

For a given state $s = (\text{history}, \text{claims})$ we also define $\text{history}(s) := \text{history}$ and $\text{claims}(s) := \text{claims}$. The set of all DFT states in a DFT \mathcal{T} is then denoted by $S(\mathcal{T})$. We also denote the empty history with $()$. To extend a history, we use the notation $\text{history}|B := \text{history} \cdot B$. Introducing a separate operator to extend a history will prove helpful later when we need to consider handling repair events. Furthermore, we define the mapping $\text{failed}: S(\mathcal{T}) \rightarrow 2^{N(\mathcal{T})}$ giving the set of all failed nodes in a given DFT state. In the same vein, define $\text{active}: S(\mathcal{T}) \rightarrow 2^{N(\mathcal{T})}$ to be a mapping giving the set of all active nodes. Furthermore, given a basic event b we let $cl(s, b) \in BES(\mathcal{T})$ denote the transitive closure of all simultaneously failed basic events due to the failure of b and the propagation of FDEPs while being in state s . Note that the dependence of the state s is necessary since certain elements of the closure might have already failed.

Let $enabled_M(s) \subseteq BES(\mathcal{T})$ further denote the set of Markovian basic events that can occur in a DFT state s . That is, $b \in enabled(s)$ iff:

- $D(b) = (\lambda, d, \mathbf{EXP})$ (Markovian event),
- $b \notin failed(s)$ (Not failed),
- $\lambda > 0$ (Can fail), and
- $b \in active(s)$ or $d > 0$ (Active or can fail dormantly).

For the transition rate of a basic event b with $D(b) = (\lambda, d, \mathbf{EXP})$ in state s , we need to consider the activation state and R-DEPs influences on the occurrence rate of b . For this we define state-dependent failure rate $\lambda(b, s) := \lambda \cdot d' \cdot y$ with:

- $d' := 1$ iff $b \in active(s)$ and otherwise $d' := d$.
- Let $R := \{r \in G(\mathcal{T}) \mid failed(r), T(\mathcal{T}, r) = x\text{-RDEP}\}$ denote the set of failed RDEPs. Then y is the total rate factor $y := \sum_{\{x \mid T(\mathcal{T}, r) = x\text{-RDEP}\}} x$.

From the formal definition of a DFT state, it is also straightforward to formalize the effect of applying a recovery action to a DFT state.

Definition 4.3 (Recovery Action Semantics). *Let r be a recovery action, and $s = (history, claims)$ a DFT state of a DFT \mathcal{T} . Then the application of r to a DFT state $s = (history, claims)$ is defined as the state $\llbracket r \rrbracket s$ with:*

- $\llbracket [] \rrbracket s := s$,
- $\llbracket CLAIM(G, S) \rrbracket s := (history, claims \cup \{(G, S)\})$, and
- $\llbracket FREE(S) \rrbracket s := (history, claims \setminus \{(G, S) \mid G \in G(\mathcal{T})\})$.

Ultimately, our goal is to define the semantics of a DFT in terms of Markov Chains to compute meaningful metrics using model checking techniques. That is, a semantics for DFTs is a mapping from a DFT to a Markov Chain.

Definition 4.4 (DFT Semantics). *A DFT semantics is a mapping $\llbracket \cdot \rrbracket$ from a DFT \mathcal{T} to an MC $\mathcal{C} = (S, T, I, L, R) = \llbracket \mathcal{T} \rrbracket$ such that:*

- $Codomain(L) \subseteq 2^{\{\mathbf{OP}, \mathbf{FAIL}\}}$,
- $\mathbf{FAIL} \in L(s)$ iff $TLE(\mathcal{T}) \in failed(s)$ for any state $s \in S$, and
- $\mathbf{OP} \in L(s)$ otherwise.

Before moving on to the NdDFT semantics, we briefly revisit the claim that merely restricting SPARE gates from claiming from left to right is not sufficient to uniquely define a unique DFT semantics. We further show that this lack of uniqueness is not merely syntactical but also has consequences for the metrics.

Proposition 4.1. *Let $\llbracket \cdot \rrbracket_{\text{DET}}$ informally denote the standard deterministic DFT to MC transformation. Then:*

- $\llbracket \cdot \rrbracket_{\text{DET}}$ is not a proper DFT semantic, and
- the MTTF metrics of the Markov Chains compatible with $\llbracket \cdot \rrbracket_{\text{DET}}$ are not equal.

Proof. To prove the claim, we give a concrete example of a fault tree with a spare race as depicted in Fig. 4.1 on page 38 where the spare race can be resolved with two equally valid Markov chains. Further, these Markov chains will have different MTTFs. Consider a DFT \mathcal{T} combining the memory system running example and the switch running example as depicted in Fig. 4.6.

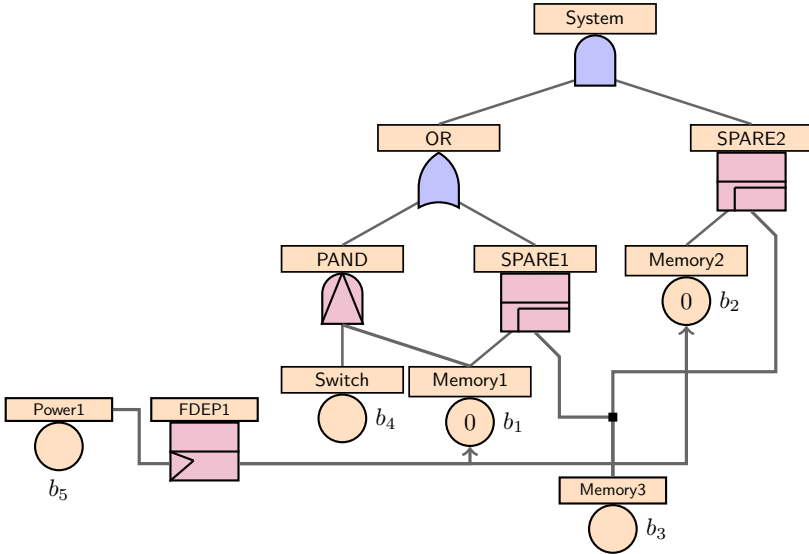


Fig. 4.6. Spare memory system variation with a switch. Different ways of resolving the spare race yield different MTTFs.

If the BE b_5 fails, a spare race occurs. Should the BE b_4 occur before b_5 , then resolving the spare race by claiming Memory3 with SPARE1 leads to a failure. Claiming Memory3 with SPARE2, however, does not lead to a failure. To keep the order in which the BEs can occur simple, let $F(b_1) = F(b_2) = 0$. Two possible Markov chains for resolving the spare race are given in Fig. 4.7.

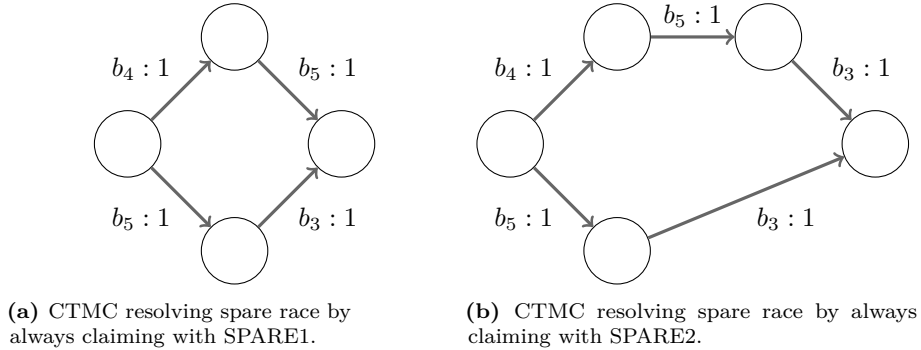


Fig. 4.7. Two CTMC options to resolve the spare race. Either by claiming with Memory1 or with Memory2. Edges are also labeled with the corresponding failed basic event.

Since both Markov chains comply to $\llbracket \cdot \rrbracket_{\text{DET}}$, it follows that $\llbracket \cdot \rrbracket_{\text{DET}}$ is not a unique mapping and therefore not a proper DFT semantics by Def. 4.4. Finally, calculating the MTTFs gives:

$$MTTF(\mathcal{C}_2) = 2 > 1.5 = MTTF(\mathcal{C}_1)$$

□

As was shown in the previous examples, the standard DFT semantics allows for the production of Markov chains that not only differ in their structure but also in essential metrics, such as the MTTF. In the NdDFT with the Markov Automaton, we will aim to obtain a proper DFT semantics that is always unique and optimal with respect to a target metric, for example, MTTF.

4.3 NdDFT with Repair

To move onto the NdDFT semantics, we need one more ingredient: To handle the ability of repair. Until now, basic events are assumed to persist permanently.

However, even when it is impossible to replace broken hardware, this assumption is not necessarily valid. Software can be reset, malfunctions caused by radiation may be transient, and operator errors might be fixed. An interesting semantic feature is therefore the introduction of **repair semantics**. Here, a basic event b can also be equipped with a repair rate $r(b)$, modeling repair processes and transient failure.

Formally, to incorporate the repair rate, we update the definition of an NdDFT with an additional distribution.

Definition 4.5. (*Repairable Fault Tree*) A Repairable Fault Tree (RFT) $\mathcal{T} = (F, B, G, T, P, D, r)$ is a tuple such that:

- (F, B, G, T, P, D) is a fault tree, and
- $r: B \rightarrow \mathbb{R}_{\geq 0}$ assigns each basic event a repair rate.

If no further specifics are given in the following, we assume all NdDFTs to be RFTs. Note that any RFT with $r(b) = 0$ for any basic event b is semantically equivalent to an FT. However, the introduction of repair events raise new semantic questions, particularly regarding SPARE gates:

- When the primary unit is repaired, should a SPARE gate switch back?
- Or should it remain with its current claim?

One deterministic approach would be to treat repair events as an inverse to failure events, canceling each other out. In this approach, a SPARE gate would therefore always revert to its original state after the prime has been repaired. However, there are many cases where this is undesirable in realistic scenarios.

For example, consider a spacecraft system with units that may suffer from wear at each activation. An example of this could be thrusters that only guarantee to operate for a limited number of activations. Consider an FDIR dependent on the temperature of the primary thruster branch that switches to the redundant unit in case of low temperature of the primary unit. In this scenario, an autonomous switch back to the primary unit once its temperature is back in the operational range is not desired, as in the case of multiple switches, it increases wear of the redundant unit, leading to its permanent failure.

A simple example fault tree illustrating the described scenario is given in Fig. 4.8 on the following page. The basic events $Fail_1$, $Fail_2$ are transient with a repair rate > 0 , whereas $Wear_1$, $Wear_2$ follow a discrete distribution. Whenever

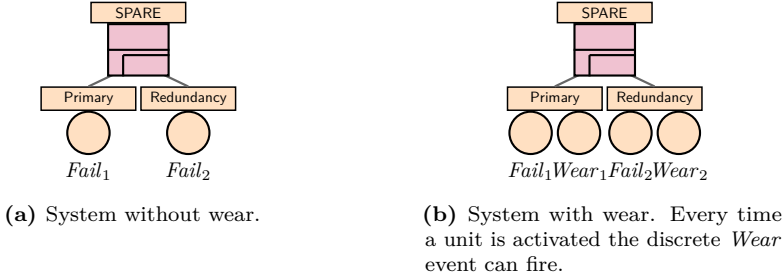


Fig. 4.8. Spare system without wear versus system with wear. Due to the Wear events, it is optimal to reduce the amount of switching between Primary and Redundancy.

primary or redundancy is activated, $Wear_1$ or $Wear_2$ may fire, respectively. To maximize the MTTF, it would therefore be desirable not to switch back.

These issues with deterministic repair semantics in DFTs are reminiscent of the known problems in deterministic DFTs, such as SPARE races. We therefore propose to resolve them similarly by introducing non-determinism into the repair semantics. By resolving the non-determinism against a desired RAMS metric, a recovery strategy describing the optimal recovery behavior with respect to the given metric can then be obtained using the previous workflow for synthesizing recovery automata.

4.3.1 FDEP with Repair

There exists, however, one type of gate that requires further special attention: The FDEP gate. Here, the following special scenarios have to be considered:

1. The trigger event fails, and the dependent event is repairable. In this case, the dependent event cannot be repaired as long as the triggering event persists.
2. The trigger event fails first and is then repaired. Then the dependent event should also be repaired.
3. The dependent event fails first, then the trigger event fails, and then the trigger event is repaired. In this case, the repair of the trigger event, contrary to the prior case, should not transitively propagate.

Taking these together, the state space semantic of a basic repair event requires taking the history into account. In the following section, this will be achieved by

defining the closure operation of repairing a basic event. The situation in Fig. 4.9 illustrates a simple power system exhibiting the different FDEP interactions. Should a transient outage occur, then we expect the system to be back online again after the outage is fixed. Should a short circuit event occur, then the system stays down permanently. The system itself has a self-repair functionality. In case it fails due to its own basic event, but during some power loss, the system cannot go back online utilizing its self-repair.

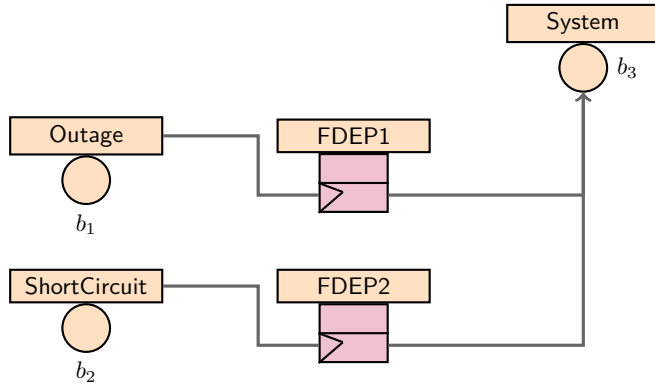


Fig. 4.9. Example constellations involving FDEP gates. The outage is transient but the short circuit is permanent. Repair rates $r(b_1) = r(b_3) = 1, r(b_2) = 0$.

4.3.2 Extended Notation with Repair

We extend the previously introduced notation to be able to also deal with repair events. For a given DFT \mathcal{T} , we extend the notation by the following symbols: For every basic event b , we introduce the respective repair event b^r . Correspondingly, for a basic event set $B = \{b_1, \dots, b_n\}$, we introduce the repair event set $B^r := \{b_1^r, \dots, b_n^r\}$. The set of all non-empty repair event sets is given by $RES(\mathcal{T})$, which mirrors the set of all non-empty basic event sets $BES(\mathcal{T})$. Likewise, $cl(s, b^r) \in RES(\mathcal{T})$ denotes the transitive closure of all basic events that are simultaneously repaired through the repair of b^r according to the conditions of Section 4.3.1 given the current DFT state s . Note that we prohibit in (1) that dependent events can be repaired if they have a failed triggering event. This guarantees $cl(s, b^r) \neq \emptyset$ for all other cases. The new set of all possible event sets is then obtained as:

$$ES(\mathcal{T}) := BES(\mathcal{T}) \cup RES(\mathcal{T})$$

Event sets for which we disregard whether they are repair events or not, we simply denote by $E \in ES(\mathcal{T})$. To extend a history $history = (B_1, \dots, B_n)$ with a repair event, we define:

$$history|B^r := (B'_1, \dots, B'_m)$$

with $B'_i := B^r \setminus B^r$ for any $1 \leq i \leq n$ and $m \leq n$, filtering out basic event sets that are empty after removing the events of B^r .

For the enabled Markovian repair events in a given DFT state s , define analogously to $enabled_M(s)$ the set $enabled_M^r(s) \subseteq RES(\mathcal{T})$. That is, $b \in enabled_M^r(s)$ iff:

- $b \in failed(s)$ (Only repair failed BEs),
- $r(b) > 0$ (BE is repairable),
- $\neg \exists g \in G(\mathcal{T}) : T(g) = \text{FDEP} \wedge g \in failed(s) \wedge (g, b) \in P(\mathcal{T})$
(Cannot repair when FDEP triggered),

4.4 Markov Automaton Semantics

With the preparational work out of the way, we can now define the semantics of an NdDFT. As discussed previously, to define the semantics, we will employ Markov automata, and in the following, we discuss how to construct an MA from an NdDFT concretely.

Note that as sketched in the transformation road map, the MA is still a non-deterministic object that will, for any fault occurrence, contain transitions for all applicable recovery actions. It only becomes possible to determinize the MA and get the final deterministic DFT semantics after an (optimal) scheduler has been computed.

Definition 4.6 (NdDFT-MA Semantics). *An NdDFT-MA semantics is a mapping from an NdDFT \mathcal{T} to an MA $\mathcal{A} = MA[\mathcal{T}]$.*

Transforming an NdDFT $\mathcal{T} = (F, B, G, T, P, D)$ into a Markov automaton

$$MA[\mathcal{T}] := \mathcal{A} = (S, L, A, N, C, P, s_0)$$

can be done by adapting traditional state space generation algorithms for transforming DFTs to CTMCs. As a base algorithm, we use the one given in [42]. The adapted algorithm will produce three types of states:

- Markovian states where outgoing transitions represent the occurrence of an exponentially distributed basic event set.
- Non-deterministic states where outgoing transitions represent the application of a recovery action.
- Probabilistic states where outgoing transitions represent the occurrence of an immediate basic event set.

The state space S will memorize the reachable DFT state information. In other words, $S \subseteq S(\mathcal{T})$. For the labeling function, we memorize the state type designation and whether a state is a fail state (top-level event has occurred). Formally, the labeling function is therefore of the form:

$$L : S \rightarrow 2^{\{\mathbf{M}, \mathbf{N}, \mathbf{P}\} \cup \{\mathbf{FAIL}, \mathbf{OP}\}}$$

As initial state, the algorithm generates the state with an empty history and no claims, i.e., $s_0 := ((), \emptyset)$. If there exists at least one basic event $b \in \text{active}(s_0)$ with $D(b) = (p, \mathbf{DISCRETE})$ for some p , then mark s_0 as a probabilistic state, that is, $L(s_0) := \{\mathbf{P}\}$. Otherwise mark it as Markovian, that is, $L(s_0) := \{\mathbf{M}\}$.

Let $s = (\text{history}, \text{claims})$ be a generated state. Add the label **FAIL** to $L(s)$ iff $\text{tle}(\mathcal{T}) \in \text{failed}(s)$. Produce the successors according to the following rules:

Markovian state successors Let s be a Markovian state with $\mathbf{M} \in L(s)$. First, we process the fault events; afterward, the repair events. Let $b \in \text{enabled}_M(s)$ be an enabled Markovian basic event failure. The basic event set B incorporating transitive failure from FDEPs is then $B := \text{cl}(s, b)$. Generate the successor:

$$s' := (\text{history}|B, \text{claims})$$

Finally, add the Markovian transition $s \xrightarrow{B:\lambda(b,s)} s'$ to C . For the repair case, proceed similarly to the fault case. For any $b^r \in \text{enabled}_M^r(s)$, let $B^r = \text{cl}(s, b^r)$. Generate the successor:

$$s' := (\text{history}|B^r, \text{claims})$$

and add the transition $s \xrightarrow{B^r:r(b)} s'$ to C . In case for any $b^r \in B^r$ it holds that $D(b^r) = (p, \text{DISCRETE})$ for some p , then set $b^r \notin L(s')$. In both cases mark the generated state as non-deterministic by adding $L(s') := L(s) \cup \{\mathbf{N}\}$.

Non-deterministic state successors Let s be non-deterministic with $\mathbf{N} \in L(s)$. Then let m be the Markovian successor state reachable with a minimal number of transitions, or s_0 if it does not exist. Let $\text{enabled}(s)$ denote the set of recovery actions, that are enabled in s . That is the following conditions hold:

- $\square \in \text{enabled}(s)$ (Empty recovery action is always enabled).
- $\text{CLAIM}(G, S) \in \text{enabled}(s)$ iff $S \notin \text{failed}(s), \neg \exists G : (G, S) \in \text{claims}$ (Cannot claim a failed or already claimed spare).
- Let $m \xrightarrow{E} s_1 \xrightarrow{r_1} \dots \xrightarrow{r_n} s$ be a path from m to s . The following conditions need to hold for the recovery action sequence $r_1 \dots r_n$:
 - If $r_i = \text{CLAIM}(G, S)$ for some G, S, i , then $\text{FREE}(S) \notin \text{enabled}(s)$ for any S . Allow CLAIM actions to be generated only after FREE actions.
 - If $r_i = \text{FREE}(S)$ for some S, i and for some SPARE gate G it holds that $(S, G) \in \text{claims}(m)$, then $\text{CLAIM}(G, S) \notin \text{enabled}(s)$. Prohibit a spare claimed in m to be freed and then claimed again by the same SPARE gate.

Furthermore, require the NdDFT gate semantic of the SPARE gate to uphold, that is, that CLAIM and FREE actions are only enabled if a spare gate was affected through an event. Formally, let G be a claiming spare gate if $r = \text{CLAIM}(G, S)$, or $(G, S) \in \text{claims}(s)$ and $r = \text{FREE}(S)$. Then there must exist a child c of G with: It either holds $c \in \text{failed}(m)$ and $c \notin \text{failed}(s)$, or $c \notin \text{failed}(m)$ and $c \in \text{failed}(s)$.

Let $r \in \text{enabled}(s)$, then generate the successor:

$$s' := \llbracket r \rrbracket s$$

and add the non-deterministic transition $s \xrightarrow{r} s'$ to N . If $r = \square$ or $\text{enabled}(s') = \emptyset$, then mark s' as probabilistic by adding $L(s') := L(s) \cup \{\mathbf{P}\}$.

Probabilistic state successors Let s be probabilistic with $P \in L(s)$. Then define $enabled(s)$ to be the set of enabled probabilistic events. That is:

- $D(b) = (p, \text{DISCRETE})$ (Immediate event),
- $b \in active(s)$ (Active),
- $b \notin failed(s)$ (Not failed), and
- s is the initial state or let u be the predecessor state of s , then $b \notin active(u)$ (Immediate event was activated through a preceding recovery action, or, if none exist, then this is the initial state).

Then, generate a successor s' for any subset $B \subseteq enabled(s)$. Let $B' = cl(B)$ be the set of failed basic events under closure of FDEPs. The probability for B' to occur is $p_B = \prod_{b \in B} p(D(b))$. Define

$$s' := (history|B', claims)$$

and add the transition $s \xrightarrow{B':p_B} s'$ to P . Mark s' as non-deterministic by setting $L(s') := L(s') \cup \{\mathbf{N}\}$.

Finally, generate the successor state $s'' = s$ for no probabilistic event triggering. For the transition probability, let $p'' = \prod_{b \in B} (1 - p(D(b)))$, and add the transition $s \xrightarrow{\emptyset:p''} s''$. Mark s'' as Markovian by adding $L(s'') := L(s'') \cup \{\mathbf{M}\}$.

This yields an MA with alternating sequences starting with a Markovian, being followed by a non-deterministic state, and finally being finished by a probabilistic state. The exception is the initial state, which may have initially active immediate basic events. The transitions are respectively labeled by the failure or repair rates of the basic events, the recovery actions, or the total occurrence probability of an immediate basic event set.

For performance purposes and to obtain a unique, minimal Markov chain, the procedure is also extended to additionally check for each generated state if there is an equivalent (i.e., probabilistically bisimilar [32]) state and reduces the state space accordingly.

4.4.1 Construction Examples

The following considers some example constructions to illustrate various cases of the Markov automaton semantics. Further examples resulting in larger state spaces can be found in Section 5.2.

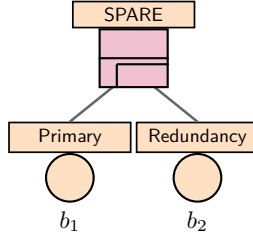


Fig. 4.10. NddFT consisting of a SPARE gate with a primary and a redundant unit. Failure rates $F_A(b_1) = F_A(b_2) = 1$, $F_D(b_1) = F_D(b_2) = 0$.

4.4.1.1 Simple NddFT to MA

As a simple example, we consider transforming the very simple NddFT shown in Fig. 4.10 to a Markov automaton. The resulting MA is given in Fig. 4.11. The active failure rates of the basic events are assumed to be 1 for b_1 and b_2 . The dormant failure rates are 0 for both events.

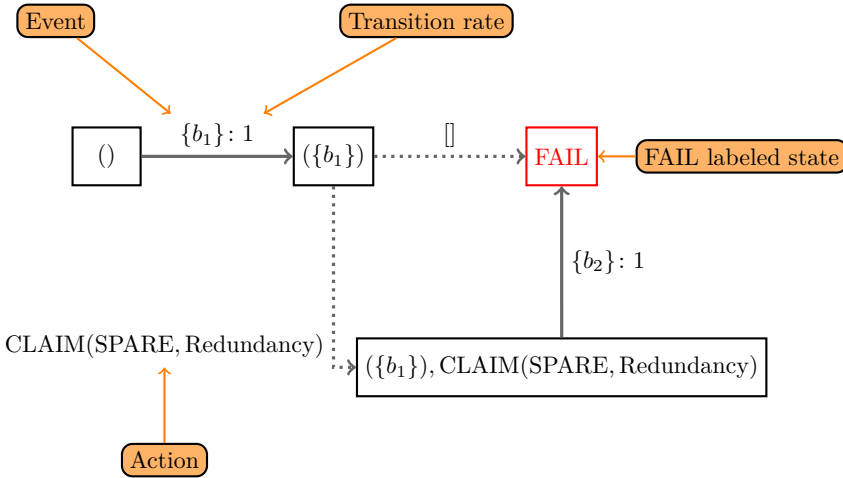


Fig. 4.11. Example transformation of an NddFT to the corresponding MA. Markovian transitions are represented with solid edges and dotted edges represent non-deterministic transitions. Since b_2 has a dormant failure rate of 0, it can only start failing after being claimed.

After the basic event set $\{b_1\}$ fails, there are two possible recovery actions:

- the empty action (\square), or
- the activation of the spare redundancy ($\text{Claim}(\text{SPARE}, \text{Redundancy})$).

Note that b_2 is initially dormant. Hence, the initial state $((), \emptyset)$ has only one successor, which is reachable by a transition labeled with $\{b_1\}: 1$. If the set of claims is empty we also simplify writing the state by dropping the set of claims and only writing the history, giving the simple initial $()$ state in the figure. Dotted edges represent the non-deterministic transitions, and solid lines represent the Markovian transitions. $B: \lambda$ denotes that the basic event set B occurs (actively or dormant) with rate λ . In this simple example, it is evident that immediately activating the redundancy upon observing $\{b_1\}$ is the correct course of action. For the FAIL labeled states, to simplify the visual depiction, we merge them into one state simply called FAIL.

4.4.1.2 NdDFT With Discrete Event to MA

We consider the effects of discrete failure events in the example NdDFT shown in Fig. 4.12.

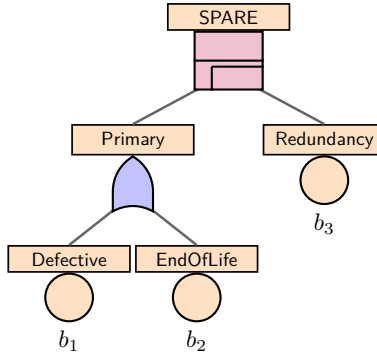


Fig. 4.12. NdDFT variation with a discrete probabilistic event. The BE b_1 models a probabilistic manufacturing failure of the Primary.

The NdDFT further breaks down the cause of failure for the Primary. The basic event b_1 models the probabilistic, discrete, and immediate failure of the Primary. On the other hand, the events b_2, b_3 model the usual exponentially distributed fault occurrence. We assume the following values:

- $D(b_1) = (50\%, \text{DISCRETE})$
- $D(b_2) = (1, \text{EXP})$
- $D(b_3) = (1, \text{EXP})$

The resulting MA is given in Fig. 4.13. The initial state is a probabilistic state with one enabled immediate event b_1 . Since the probability of b_1 occurring is 50%, the probability of no basic event occurring is 50%, hence giving the \emptyset : 50% transition. Note that if no event occurs, the history remains as $()$, however, the successor state is labeled as a Markovian state.

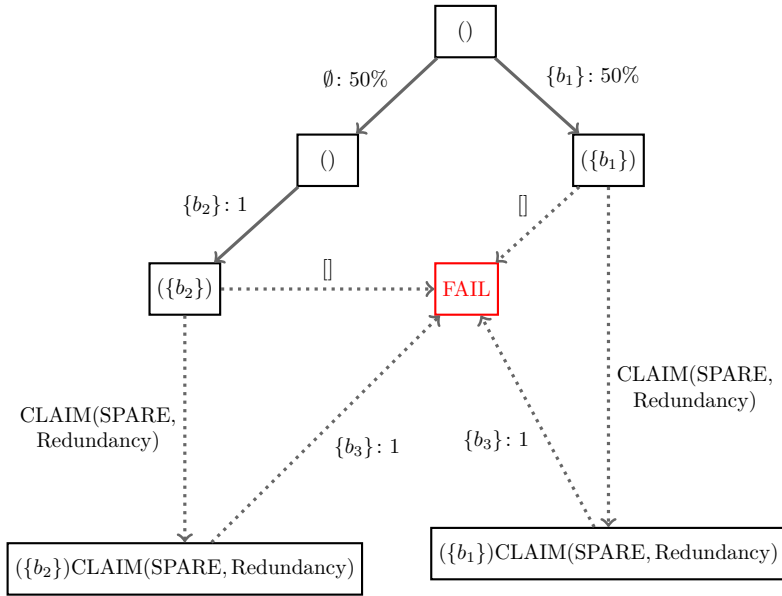


Fig. 4.13. Example transformation of an NdDFT with a discrete event to the corresponding MA. The left path from the initial state represents the discrete event not occurring, the right path represents its occurrence.

The states with a **FAIL** label have been immediately merged into one single state called **FAIL** for improved visual clarity. Note that $(\{b_2\})$ and $(\{b_1\})$, as well as $(\{b_2\})\text{CLAIM}(\text{SPARE}, \text{Redundancy})$ and $(\{b_1\})\text{CLAIM}(\text{SPARE}, \text{Redundancy})$ are bisimilar. While the formal semantics produces them as distinct states, an implementation merging bisimilar states would merge them.

4.4.2 Repairable NdDFT to MA

For a repairable example, we reconsider the simple DFT from Fig. 4.10 on page 55 but equip the basic event b_1 with a repair rate $r(b_1) = 1$. The basic event b_2 , on the other hand, remains non-repairable. Even though we only employ a single repair rate, we will see in this example how the repair process can significantly blow-up the MA state space. In the following, the name of the node SPARE is abbreviated with S and the name of the node Redundancy by R. The resulting MA is shown in Fig. 4.14.

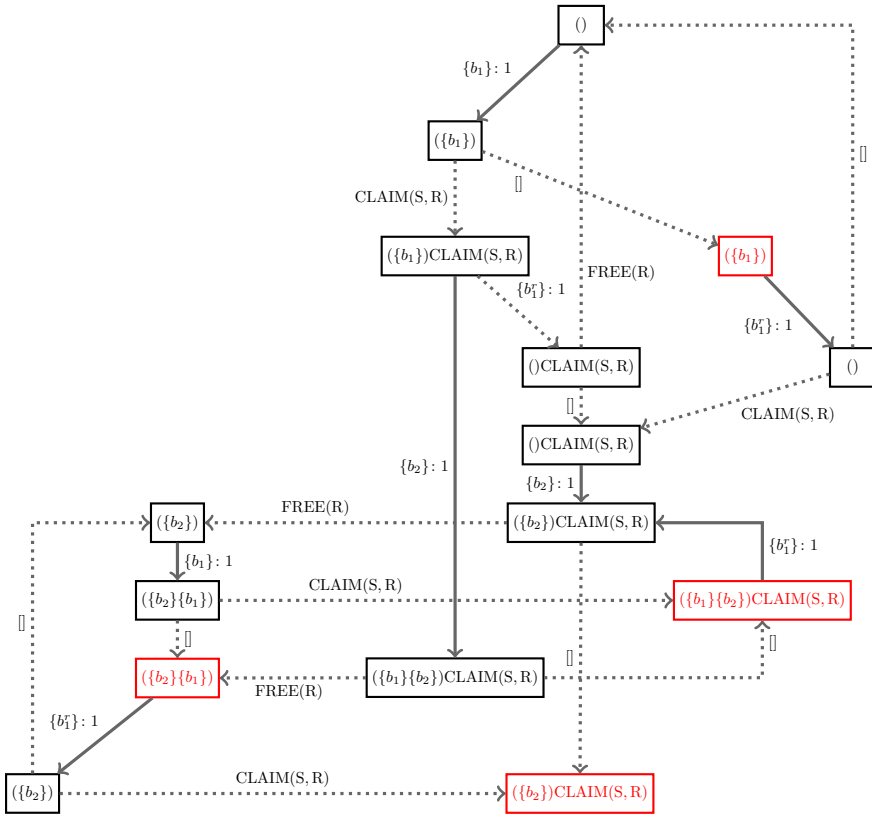


Fig. 4.14. Example transformation of a repairable NdDFT to the corresponding MA. The basic event b_1 is repairable while b_2 is non-repairable.

Symmetric states of the form $(\{b_1\}, \{b_2\})$ and $(\{b_2\}, \{b_1\})$ have already been merged to simplify the visual depiction. Once b_1 has failed, the Markovian repair event $\{b_1'\}$ becomes enabled. Due to b_1 being repairable, new, previously unreachable states, such as $(\{b_2\})\text{CLAIM}(S, R)$ become reachable, which then cascade into further newly reachable successor states.

4.4.3 Recovery Strategies and Automata

To resolve the non-determinism present in NdDFTs, the actual recovery actions to be applied in failure cases are given by recovery strategies implemented by recovery automata. In the following, transitions of recovery automata are labeled by a triggering set of basic events - either denoting failure or repair - and a recovery action sequence. The elements of the triggering set are also called *guards*. Moreover, we extend the definition of recovery actions to the set of recovery action sequences by

$$RS(\mathcal{T}) = (R(\mathcal{T}) \setminus \{\emptyset\})^*$$

For recovery action sequences, the empty action is ignored and considered as the empty word ϵ . Given the observed basic events, a recovery strategy is then a mapping that returns the recovery action sequence that should be taken accordingly.

The NdDFT considers recovery strategies that are composed of recovery actions as given in Def. 4.1. They are defined as follows:

Definition 4.7 (Recovery Strategy). *A recovery strategy for an NdDFT \mathcal{T} is a mapping $\text{Recovery} : ES(\mathcal{T})^* \rightarrow RS(\mathcal{T})^*$ such that*

- $\text{Recovery}(\epsilon) = \epsilon$ and
- $\text{Recovery}(E_1, \dots, E_n) = \text{Recovery}(E_1, \dots, E_{n-1}), rs_n$ with $rs_n \in RS(\mathcal{T})$.

A recovery strategy that could be synthesized for the example MA depicted in Fig. 4.11 on page 55 in this example would thus yield $\text{Recovery}(\{b_1\}) = \text{Claim}(\text{SPARE}, \text{Redundancy})$.

To represent recovery strategies, we will use automata. A finite automaton that represents a recovery strategy will be called *recovery automaton*. Formally, a recovery automaton is a Mealy machine [66] having the event sets $ES(\mathcal{T})$ as the input alphabet and the set of recovery action sequences $RS(\mathcal{T})$ as the output alphabet. This concept is formalized in the following:

Definition 4.8 (Recovery Automaton). *A Recovery Automaton (RA) $\mathcal{R}_{\mathcal{T}} = (Q, \delta, q_0)$ of an NddFT \mathcal{T} is an automaton where*

- Q is a finite set of states,
- $q_0 \in Q$ is the initial state, and
- $\delta: Q \times ES(\mathcal{T}) \rightarrow Q \times RS(\mathcal{T})$ is a deterministic transition function that maps the current state and an observed set of faults, or repairs, to the successor state and a recovery action sequence.

The transition function δ is extended to $\delta^*: Q \times ES(\mathcal{T})^* \rightarrow RS(\mathcal{T})^*$ by letting

$$\begin{aligned} \delta^*(q, \epsilon) &:= \epsilon \\ \delta^*(q, E \cdot w) &:= rs \cdot \delta^*(q', w) \text{ with } \delta(q, E) = (q', rs) \end{aligned}$$

for any $q \in Q$, $E \in ES(\mathcal{T})$ and $w \in ES(\mathcal{T})^*$. The recovery strategy induced by a recovery automaton \mathcal{R} , $Recovery_{\mathcal{R}}: ES(\mathcal{T})^* \rightarrow RS(\mathcal{T})^*$, is given by $Recovery_{\mathcal{R}}(w) := \delta^*(q_0, w)$.

Note that the set of states, Q , is not further specified in our formal definition of RA. In Chapter 5, we will see how it can be obtained from the Markov automaton of an NddFT after computing an optimal scheduler. As the internal structure of those states (which record the history of error events that occurred and the assignment of spare components) is not relevant for further optimizations, we will use symbolic states of the form q_i in our examples. In the RAs depicted in the following, if no transition is explicitly defined for some state q and some input E , then it is assumed to be an ϵ -loop transition, i.e., $\delta(q, E) = (q, \epsilon)$.

An example of a recovery automaton for a simple fault tree consisting of a SPARE gate with a cold redundant spare is given in Fig. 4.15. If the primary unit fails, the SPARE gate switches to the redundancy unit by claiming it.



Fig. 4.15. Example RA for the simple NddFT from Fig. 4.10 on page 55. When b_1 occurs, the Redundancy is claimed. When b_2 occurs, the recovery strategy is empty.

By composing the semantics of the recovery automaton with the Markov automaton semantics of an NddFT, we can now formally define the deterministic

DFT semantics of an NdDFT by means of Markov chains. The construction involves some formal detail but is straightforward. The key idea is to synchronize the non-deterministic transitions in the MA with the deterministic transitions in the RA. There is one technical issue that complicates the composition: Encoding the probabilistic transitions of the Markov automaton in the Markov chain. Markov chains only possess one type of transitions, in this case, as we are employing continuous-time Markov chains, these are transitions with exponential distributions. However, we also need to encode the Markov automaton's discrete, probabilistic transitions. While not demanding in terms of algorithmic complexity, this causes some notational blowup as the Markov automaton may contain several subsequent paths with immediate events.

Definition 4.9 (NdDFT-RA Composition). *An NdDFT-RA composition is a function \parallel that maps an NdDFT \mathcal{T} and an RA $\mathcal{R}_{\mathcal{T}}$ to a Markov chain*

$$\mathcal{C} := \mathcal{T} \parallel \mathcal{R}_{\mathcal{T}}$$

Let $\text{MA}[\mathcal{T}] = (S, L, A, N, C, P, s_0)$ be the Markov automaton of \mathcal{T} . Further, let $\mathcal{R}_{\mathcal{T}} = (Q, \delta, q_0)$ be a recovery automaton. Then construct the continuous time Markov chain $\mathcal{T} \parallel \mathcal{R} := \mathcal{C} = (S', T', I', L', R')$ as follows.

We encode the current state of the recovery automaton in the state space, that is, $S' \subseteq S \times Q$. For the labeling function L choose:

$$L' : S \rightarrow 2^{\{OP, FAIL\}}$$

If s_0 is a Markovian state with $M \in L(s_0)$, then generate the initial state $s'_0 := (s_0, q_0)$ by setting $I'(s'_0) = 1$ and $I'(s') = 0$ for any other state s' . An immediate probabilistic transition occurring in the initial state s_0 can be encoded in the initial distribution I' as follows. If s_0 is probabilistic, then consider any maximal path

$$s_0 \xrightarrow{E^0:p^0} s_1^0 \xrightarrow{r_1^0} s_2^0 \dots \xrightarrow{r_{n_0}^0} s_{n_0}^0 \xrightarrow{E^1:p^1} s_1^1 \xrightarrow{r_1^1} s_2^1 \dots \xrightarrow{r_{n_m}^m} s_{n_m}^m$$

in \mathcal{A} such that for any $M \in L(s_{n_m}^m)$ and $P \in L(s_{i_k}^j)$ for any other state $s_{i_k}^j$. If there is a path

$$q_0 \xrightarrow{E^0:r_1^0 \dots r_{n_0}^0} q^1 \dots \xrightarrow{E^m:r_1^m \dots r_{n_m}^m} q^m$$

in the recovery automaton \mathcal{R} , then generate the state $(s_{n_m}^m, q^m)$ with $I((s_{n_m}^m, q^m)) = \prod_{i=1}^m p^i$. Also, if $E^m = \emptyset$ and there is a path

$$q_0 \xrightarrow{E^0:r_1^0 \dots r_n^0} q^1 \dots \xrightarrow{E^{m-1}:r_1^{m-1} \dots r_{n_{m-1}}^{m-1}} q^{m-1}$$

generate the state $(s_{n_{m-1}}^{m-1}, q_{m-1})$ with $I'((s_{n_{m-1}}^{m-1}, q_{m-1})) = \prod_{i=1}^{m-1} p^i$.

Let (s, q) be a generated state. Define:

$$L(s) := (L(s) \cap \{OP, FAIL\})$$

Then generate the successors as follows. In order to encode the probabilistic transitions in the exponential ones, proceed similarly to the construction of the initial states. Consider any maximal paths of the form

$$s \xrightarrow{E:\lambda} s_1 \xrightarrow{r_1} s_2 \dots \xrightarrow{r_n} s_n \xrightarrow{E^0:p^0} s_1^0 \xrightarrow{r_1^0} s_2^0 \dots \xrightarrow{r_{n_0}^0} s_{n_0}^0 \xrightarrow{E^1:p^1} s_1^1 \xrightarrow{r_1^1} s_2^1 \dots \xrightarrow{r_{n_m}^m} s_{n_m}^m$$

in \mathcal{A} such that for any $M \in L(s_{n_m}^m)$, $P \in L(s_{i_k}^j)$ for any state $s_{i_k}^j$, and $N \in L(s_i)$ for any state s_i . If there is a path

$$q_0 \xrightarrow{E:r^1 \dots r^n} q' \xrightarrow{E^0:r_1^0 \dots r_{n_0}^0} q^1 \dots \xrightarrow{E^m:r_1^m \dots r_{n_m}^m} q'^m$$

in the recovery automaton \mathcal{R} , then generate the state $(s_{n_m}^m, q'^m)$. Also, let $p = \prod_{i=1}^m p^i$ and generate the transition $(s, q) \xrightarrow{\lambda \cdot p} (s_{n_m}^m, q'^m)$ in T' . Also, if $E^m = \emptyset$ and there is a path

$$q_0 \xrightarrow{E^0:r_1^0 \dots r_n^0} q^1 \dots \xrightarrow{E^{m-1}:r_1^{m-1} \dots r_{n_{m-1}}^{m-1}} q^{m-1}$$

generate the state $(s_{n_{m-1}}^{m-1}, q_{m-1})$. Also, let $p = \prod_{i=1}^{m-1} p^i$ and generate the transition $(s, q) \xrightarrow{\lambda \cdot p} (s_{n_{m-1}}^{m-1}, q_{m-1})$ in T' .

We now have a transformation from NdDFT to Markov automata and an operator to synchronize a recovery automaton with a Markov automaton to a Markov chain. In order to define a direct semantics of an NdDFT, the remaining ingredient is a criterion to select the appropriate recovery automaton to synchronize with. The core idea of NdDFTs was to use a recovery automaton that maximizes a given RAMS metric. Therefore, the desired semantics will depend on a given evaluation metric $M: \mathcal{C} \mapsto \mathbb{R}_{>0}$ that maps an MC \mathcal{C} to a non-negative, real value of $\mathbb{R}_{>0}$.

Definition 4.10 (NdDFT-MC Semantics). *Let M be a metric. An NdDFT-MC semantics is a mapping $MC \llbracket \cdot \rrbracket^M$ that maps an NdDFT \mathcal{T} to a Markov chain $\mathcal{C} := MC \llbracket \mathcal{T} \rrbracket^M := \mathcal{T} \parallel \mathcal{R}_{\mathcal{T}, max}$ with:*

$$\mathcal{R}_{\mathcal{T}, max} = \operatorname{argmax}_{\mathcal{R}_{\mathcal{T}}} \{M(\mathcal{T} \parallel \mathcal{R}_{\mathcal{T}})\}$$

Furthermore, we limit ourselves to positionally determined metrics M on a Markov Automaton. A metric M is said to be positionally determined iff the metric can be purely determined by the current state. In other words, if S is the state space, then a positional metric on the state level can be expressed as a mapping $M: S \mapsto \mathbb{R}_{>0}$ assigning each state its metrical value. Examples for such metrics are long-run properties such as MTTF or SSA. Not positionally determined is for example the “reliability after time t ” metric, which can yield optimal strategies that depend not only on the current state but also on the current point in time. In order to obtain a unique mapping, we also need to deal with the possibility of multiple recovery automata with different recovery behavior but yielding the same metrical evaluation. Consider for example a simple NdDFT with a SPARE gate and two redundancies with the same failure rates. Picking either results in a different recovery automaton but yields the same MTTF. Likewise, consider an NdDFT with a SPARE gate where the spare has dormant failure; switching to the spare upon the primary failure or not both result in a fail state. We therefore introduce the constraint of *action minimality* to ensure the uniqueness of our NdDFT semantics.

Definition 4.11 (Action Minimality). *Define the total order $<_{\subseteq} RS(\mathcal{T}) \times RS(\mathcal{T})$ over recovery action sequences such that:*

- $|rs_1| < |rs_2| \implies rs_1 < rs_2$, and
- if $|rs_1| = |rs_2|$ then $rs_1 < rs_2$ iff rs_1 is smaller than rs_2 according to the alphabetical order over the names of the recovery actions.

If M is positional, then there exists one unique strategy where for each DFT state, we take the action minimal recovery action to maximize the positional metric M . Hence, $\mathcal{R}_{\mathcal{T}, \max}$ is guaranteed to be unique, and therefore $MC[\mathcal{T}]^M$ is a proper mapping and therefore overall a DFT semantics according to Def. 4.4. Formally, we obtain:

Proposition 4.2. *Let M be a positional metric. Then $MC[\mathcal{T}]^M$ is a DFT semantics.*

Chapter 5

Synthesis of Recovery Strategies

The previous chapter established the necessary semantics for defining the Markov automaton for an NdDFT and synchronizing an RA with a Markov automaton. In this chapter, the focal point will be on the missing piece: Automatically generating a recovery automaton from a Markov automaton, which we refer to as the *synthesis process*. As for the actual optimization procedure for obtaining the optimal decision process in a Markov automaton, due to the established foundations, we will be able to leave the actual optimization process to a model checker. However, to extract the results, make them more usable for practical purposes, and enhance scalability, the simple previously presented workflow will be extended by several building blocks. Besides the actual synthesis, in this chapter, we discuss techniques for reducing the state space of synthesized recovery automata, going beyond commonly known reduction techniques for deterministic finite automata. Furthermore, we discuss a modularization technique to mitigate the state space explosion problem in the Markov automaton generation.

5.1 Synthesis Methodology

Using existing techniques for optimizing the scheduling of a Markov automaton, the choice of non-deterministic transitions that maximize system reliability can be computed. The recovery automata model is then used to represent the underlying decision process of the scheduler.

While the concrete metric itself is interchangeable, we focus on optimizing with regard to quantitative metrics in this work. In particular, we concentrate on the MTTF. Compared to the “reliability after time t ” metric, which is not positionally determined, the long-run MTTF metric gives the advantage of dropping the time parameter t and allows us to base the optimal scheduler purely on the current state.

More formally, as already established in the previous chapter, we limit ourselves to positionally determined metrics on a Markov automaton. Given the state space S , the optimal scheduler can then be represented as a mapping $\sigma: S \rightarrow S$, assigning the optimal successor state to each state in the state space.

For the Markov automaton, maximizing the MTTF corresponds to maximizing the expected long-term reachability property of a FAIL labeled state.

5.1.1 Extraction

With the established foundations, the actual extraction process of the recovery automaton is straightforward. The key idea is to use a scheduler provided by a model checker to resolve the non-deterministic transitions. Transitions that do not match with the scheduler are discarded. The individual recovery actions are contracted into a recovery sequence, and the labels on the Markovian and probabilistic transitions are used as the guards. Formalizing this process gives us the following:

Definition 5.1. *Define the RA $\mathcal{R}_{\mathcal{T}}$ extracted from an MA $MA[\mathcal{T}]$ as*

$$\mathcal{R}_{\mathcal{T}} := R(MA[\mathcal{T}])$$

Let $\mathcal{A} := MA[\mathcal{T}]$ be a Markov automaton for an NdDFT and have state space S . Let further $\sigma: S \rightarrow S$ be the optimal scheduler. Then, extracting a recovery automaton $\mathcal{R} := (Q, \delta, q_0) = R(MA[\mathcal{T}])$ from the scheduler σ and the Markov automaton \mathcal{A} is achieved by considering sequences of transitions for states s_0, s_1, \dots, s_n of the form:

$$s_0 \xrightarrow{E:\lambda} s_1 \xrightarrow{r_1} s_2 \xrightarrow{r_2} \dots s_{n-1} \xrightarrow{r_{n-1}} s_n,$$

where E is an event set, λ a failure rate and r_1, \dots, r_n are recovery actions. If $\sigma(s_i) \neq s_{i+1}$, for some $1 \leq i \leq n$, then the transition sequence is not optimal and thus discard it. Otherwise, the transition sequence is optimal and therefore generate the transition:

$$\delta(s_0, E) := (s_n, r_1 \dots r_n),$$

where empty recovery actions are ignored. Observe that multiple recovery actions from the Markov automaton are combined into one recovery action sequence in the recovery automaton. This applies to all transitions where s_1, \dots, s_n are the successors computed by the optimized schedule of the Markov automaton. If a state s in the MA does not define a transition for some input E , then in the recovery automaton generate the transition $\delta(s, E) := (s, \epsilon)$. Finally, the algorithm discards all unreachable states.

5.2 Examples

In the following, we consider some examples to illustrate the Markov automaton construction and recovery automaton synthesis. To this end, we give some example NdDFTs, a fragment of the Markov automaton, the synthesized RA, and some data on the reliability of the NdDFT changes through the usage of an optimized RA.

5.2.1 Construction of an Adaptable Recovery Strategy

As the first example NdDFT, we consider a model where the recovery strategy can be improved by adapting to the occurrence of some basic events. Consider the model shown in Fig. 5.1 on the next page, which is similar to the one depicted in Fig. 2.10a on page 22. Except, in this case, the NdDFT has been copied, and now two equipment are dependent on respective switches activating one shared spare. The spare Redundancy is shared among the two spare gates, SPARE1 and SPARE2. Should both subsystems fail, then the entire system fails.

For the failure rates, we consider the (unitless) values:

- $F(b_2) = F(b_5) = F_A(b_3) = 5$ (equal active failure rates for all equipments),
- $F_D(b_3) = 0$ (the spare is a cold redundancy) and
- $F(b_1) = F(b_4) = 0.1$ (low switch failure rate).

Fig. 5.2 on page 69 showcases a small excerpt of the constructed Markov Automaton limited to the fault sequence $(\{b_1\}, \{b_2\}, \{b_5\})$, and the enabled recovery actions. Since the switch b_1 is not a child of a spare gate, the only available recovery action is \square . After the occurrence of the primary b_2 , the MA faces two possible decisions: claiming the redundancy or not performing a recovery action. In the case of the chosen fault sequence $(\{b_1\}, \{b_2\}, \{b_5\})$, claiming will lead to the FAIL state.

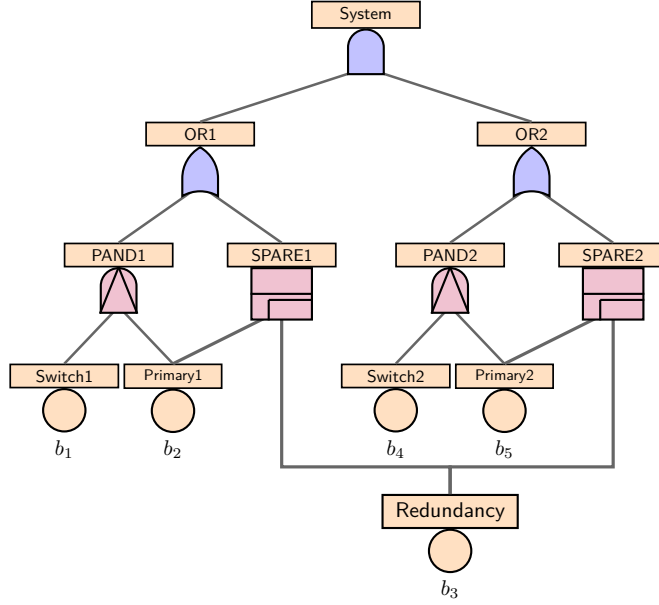


Fig. 5.1. DFT with a shared redundancy and two switches. Assigning the Redundancy to a SPARE gate after the associated Switch event occurred, does not recover the sub-tree. However, the Redundancy is then unavailable for the other SPARE gate.

Fig. 5.3 on the following page depicts the synthesized recovery automaton. We can find the corresponding fragment in the state sequence q_0, q_1, q_3 . Note that q_2 only has a non-empty recovery action defined for the occurrence of b_2 . This reflects the optimality of choosing the empty recovery action \square in order to avoid the FAIL state from the earlier discussed MA fragment. In essence, the synthesized recovery Automaton states that the spare gates SPARE1, SPARE2 should not allocate the redundancy if the respective switch has already failed.

Fig. 5.4 on page 70 shows the reliability curves for the two semantics. The x-marked line shows the reliability curve when considering the fault tree in Fig. 5.1 using the default DFT semantics, i.e., always claim. The circle-marked line shows the reliability when using the NdDFT semantics while employing the recovery automaton from Fig. 5.3 on the following page. For the time frame, a unitless mission time of 1 has been chosen. We can see that the reliability curves of both fault trees converge towards 0. However, employing an adaptive strategy

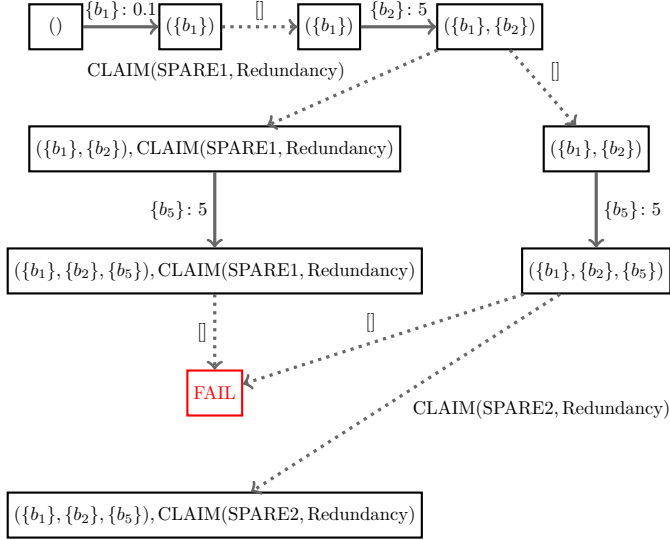


Fig. 5.2. Excerpt from the Markov Automaton for the double switch NddFT for the fault sequence $(\{b_1\}, \{b_2\}, \{b_5\})$. The correct action is to not claim after b_2 occurs to preserve the spare.

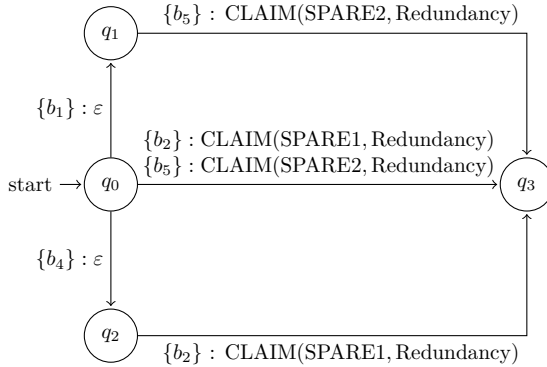


Fig. 5.3. Synthesized recovery automaton for the switch system. The occurrence of b_1 or b_4 change the recovery behavior.

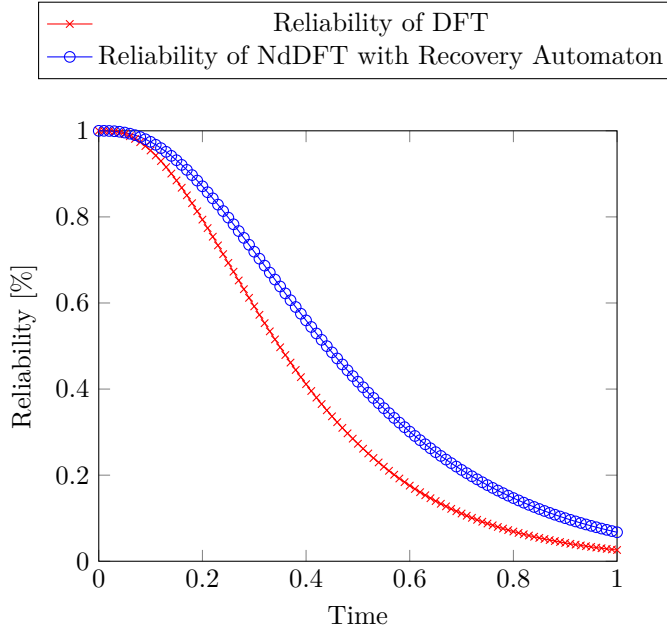


Fig. 5.4. Reliability of DFT vs. reliability of NdDFT with Recovery Automaton

Metric	DFT	NdDFT	Factor
MTTF	0.38	0.47	1.24
#States	109	149	1.43
#Transitions	146	226	1.55

Table 5.1: DFT vs. NdDFT with Recovery Automaton

for activating spares yields a reliability curve that is consistently better than the fixed order strategy of standard DFT.

Further data on the difference between the two semantics is given in Table 5.1. The improvement in reliability prolongs the mean time to failure by about 24%. On the other hand, applying the NdDFT semantics increases state space size and transition count of the corresponding MA by 43% and 55%, respectively.

5.2.2 Optimized Spare Ordering

As a second example use case, we reconsider the redundant memory system shown in Fig. 5.5.

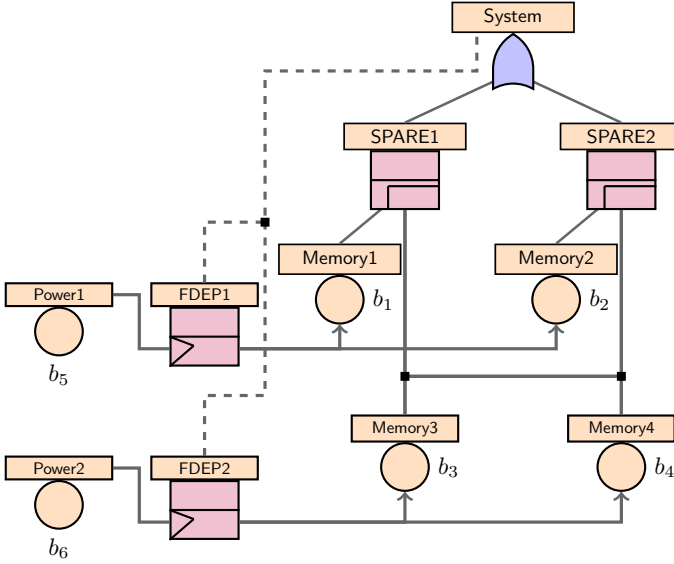


Fig. 5.5. NddFT memory system with two spares in the pool.

With the default DFT semantics, Memory3 would always be employed before Memory4. However, as will be shown in the following, depending on the failure rates, this might yield a sub-optimal strategy. Consider for example as failure rates:

- $F(b_1) = F(b_2) = 1$,
- $F_A(b_3) = 5$ (modeling a low quality spare),
- $F_A(b_4) = 0.5$ (modeling a high quality spare),
- $F_D(b_3) = F_D(b_4) = 0$ (the spares are cold redundancies), and
- $F(b_5) = F(b_6) = 0.1$ (modeling always active power sources with low failure rate).

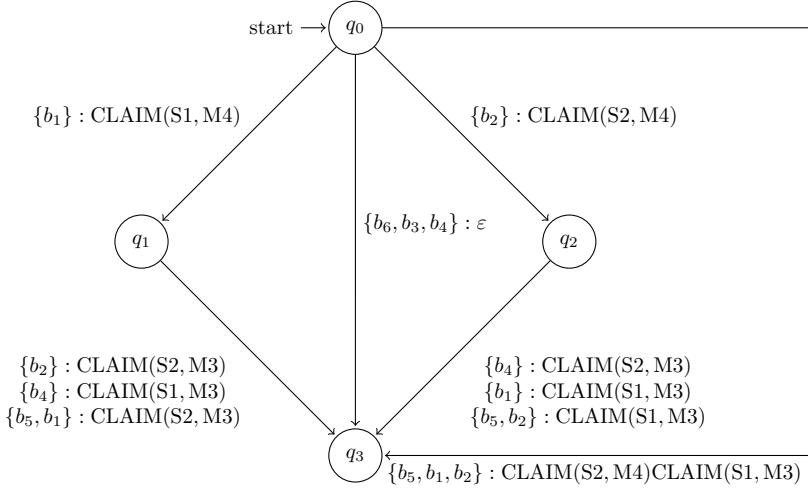


Fig. 5.6. Synthesized Recovery Automaton for memory system

Hence, according to DFT semantics, the low-quality spare will always be activated first.

For improved readability, in the following figures, the items SPARE1, SPARE2, and Memory1, . . . , Memory4 will be abbreviated by S1, S2, and M1, . . . , M4 respectively. The synthesized Recovery Automaton is depicted in Fig. 5.6.

It basically states that the system should always first activate Memory4, that is, the high-quality spare, before the low-quality spare Memory3. To evaluate the recovery strategy induced by the Recovery Automaton, the reliability curves of the fault tree models are plotted in Fig. 5.7 on the following page.

It can be seen that employing the strategy proposed by the synthesized Recovery Automaton yields a slight edge over the reliability curve of the standard DFT. In this simple example, the deterministic DFT model could yield the same performance by correcting the spare ordering. However, determining the optimal spare ordering can become exceedingly difficult as the complexity of spares, which may also be modeled by complex fault trees, increases. Additional data on the details is given in Table 5.2.

The improvement of the reliability curve yields a slight prolongation of mean time to failure by about 9%. The transition count suffers a significant increase by about 80%.

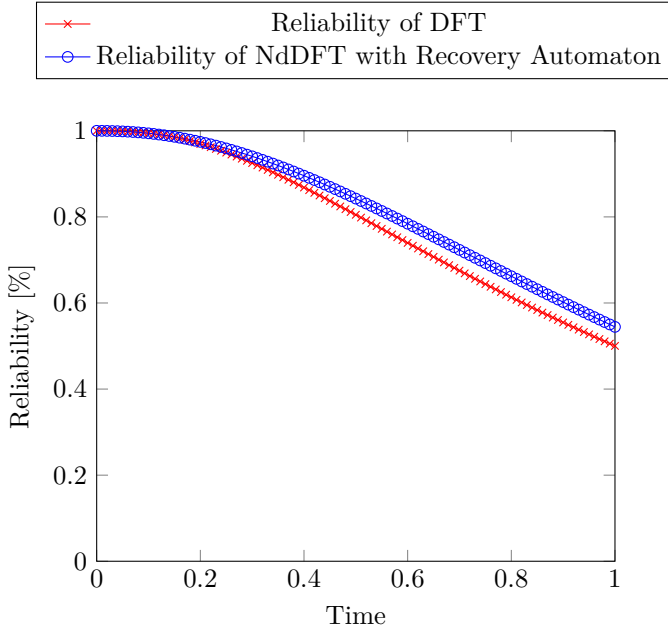


Fig. 5.7. Reliability of DFT vs. reliability of NdDFT with recovery automaton for memory system.

Metric	DFT	NdDFT	Factor
MTTF	1.09	1.19	1.09
#States	18	24	1.33
#Transitions	54	97	1.8

Table 5.2: DFT vs. NdDFT with Recovery Automaton for memory system.

5.3 Further Optimization of Recovery Automata

Complex systems usually exhibit a large number of faults that may occur, see Sec. 8.3 on page 154. This means that NdDFTs describing such systems may be very large. In this section, we refine the given synthesis procedure by discussing further techniques for reducing the state space and the transition count of a synthesized recovery automaton. This leads to the task of finding an automaton with the same “behavior” that exhibits a smaller number of states.

In the following, we will exploit the property that non-repairable faults can only occur at most once to perform a state space reduction procedure that goes further than standard automaton minimization techniques. We express a fault only occurring once in a sequence of basic event sets B_1, \dots, B_n through the condition $B_i \cap B_j = \emptyset$ for any $i \neq j$, which prevents a basic event from occurring in multiple basic event sets. Since the technique only applies to non-repairable events, we therefore restrict ourselves in this section to only non-repairable NdDFTs.

To capture the notion of two recovery automata exhibiting the same behavior, we introduce the concept of *recovery* equivalence between recovery automata as follows:

Definition 5.2 (RA Recovery Equivalence). *Let $\mathcal{R}_1 = (Q_1, \delta_1, q_{01})$ and $\mathcal{R}_2 = (Q_2, \delta_2, q_{02})$ be two RAs. We define a binary relation \sim_R such that it holds true that $\mathcal{R}_1 \sim_R \mathcal{R}_2$ iff for any sequence of sets of basic events B_1, \dots, B_n with $B_i \cap B_j = \emptyset$ for any $i \neq j$ it holds that:*

$$\text{Recovery}_{\mathcal{R}_1}(B_1, \dots, B_n) = \text{Recovery}_{\mathcal{R}_2}(B_1, \dots, B_n).$$

Given a recovery automaton as an input, the task of minimization means to compute an equivalent recovery automaton with as few states as possible. The standard problem of automata minimization is well-known and has been studied extensively. In this work, we apply the usual definition of trace equivalence and lift it to states of recovery automata:

Definition 5.3 (Trace Equivalence). *Let $\mathcal{R}_{\mathcal{T}} = (Q, \delta, q_0)$ 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 $B \in \text{BES}(\mathcal{T})$ it holds that:*

$$\delta(q_1, B) = (q'_1, rs_1) \text{ and } \delta(q_2, B) = (q'_2, rs_2) \text{ with } q'_1 \approx q'_2 \text{ and } rs_1 = rs_2$$

Equivalent states in an automaton can be computed using a partition refinement algorithm [67]. It operates by iteratively partitioning states so that states in different partitions are guaranteed to be inequivalent. On automata with acceptance conditions, this is usually achieved by using final and non-final states as the initial partitions. In this setting, the initial partitions are created by grouping states with the same input-output mappings into a partition. The algorithm then refines the partitions by identifying sub-partitions with different

output behavior. When the algorithm terminates, each partition contains equivalent states only. A minimized automaton can then be obtained by merging all equivalent states. In the setting of recovery automata, we can go even further and merge pairs of states that are not trace-equivalent as long as the behavior of the automaton does not change. A simple example for a case where merging non-equivalent states yields a recovery automaton that induces an equivalent recovery strategy can be seen in Fig. 5.8.

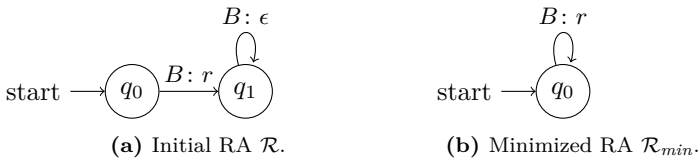


Fig. 5.8. Example for merging non-trace-equivalent states. Since B can only occur once, the transition $B : \epsilon$ can never be taken.

In the example, the states q_0 and q_1 are clearly not trace-equivalent since q_0 outputs r upon reading B whereas q_1 outputs ϵ upon reading B . However, it can be shown that the two automata are recovery-equivalent: Since B is the only input, the traces fulfilling the condition of having no input repetition according to Def. 5.2, i.e., B_1, \dots, B_n with $B_i \cap B_j = \emptyset$ for any $i \neq j$, are exactly B and ϵ . Furthermore, it holds that $Recovery_{\mathcal{R}}(\epsilon) = \epsilon = Recovery_{\mathcal{R}_{min}}(\epsilon)$ and $Recovery_{\mathcal{R}}(B) = r = Recovery_{\mathcal{R}_{min}}(B)$ by Def. 4.8. The equivalence of the two recovery automata then follows by Def. 5.2. Intuitively, the transition $B : \epsilon$ in the initial RA can never be taken due to B being disabled, as it must have already occurred. In the following, we present the main contribution of this section: Rules that optimize the state space beyond merging trace-equivalent states yet yield implementations of equivalent recovery strategies. We identify two cases where we can optimize the state space without affecting the induced recovery strategy.

- **Case 1: Optimizing Orthogonal States.**
- **Case 2: Optimizing the FAIL state.**

In both cases, the key to minimization that we exploit is that a non-repairable FT produces the inputs of the automaton. Hence, non-repairable basic events can only occur at most once. This leads to the effect that certain traces in the RA are invalid inputs for the induced recovery strategy. Therefore, this

restriction gives us additional freedom to merge states that would not be merged in a standard automata model. The updated workflow with the integrated state space reduction for the recovery automaton is given in Fig. 5.9.

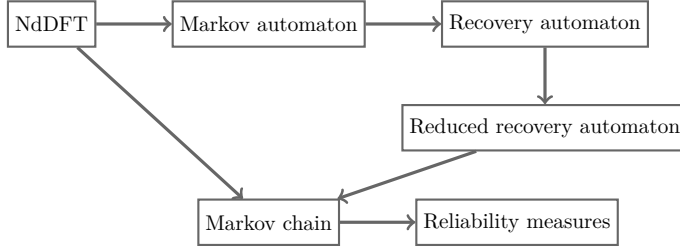


Fig. 5.9. Updated transformation workflow. Before composing NdDFT and RA, we first compute a reduced recovery automaton.

5.3.1 Optimizing Orthogonal States

In the first rule, the goal is to identify states that may have transitions with disagreeing outputs but where we can guarantee that conflicts are excluded, as their necessary inputs can no longer be produced. The key to this idea lies in exploiting the property that basic events can only occur at most once in an FT. This leads to the following observation: If a basic event occurs on every path leading to a state in an RA, then it is guaranteed that in the future, no transition listing this basic event in its guards can be taken. Note that since recovery automata are deterministic, they always have a transition defined for every possible input. Fig. 5.10 on the following page abstractly illustrates how this observation can be exploited to merge non-trace-equivalent states.

In order to reach q_1 , the event set B_2 must occur. Therefore, upon reaching q_1 , the transition $B'_2: z_2$ can no longer be taken. Similarly, the transition $B'_1: z_1$ cannot be taken in q_2 . Hence, the states q_1 and q_2 can be safely merged without changing the recovery-equivalence of the automaton. We now introduce the concept of *orthogonal states* to formalize this notion. To capture the basic event sets that can no longer be produced by an FT, we define the set of *disabled inputs* of a state q as a function $DI: Q \rightarrow 2^{BES(\mathcal{T})}$ with:

$$DI(q) := \{B \in BES(\mathcal{T}) \mid \text{for all paths } q_0 \xrightarrow{B_0: r^{s_0}} q_1 \xrightarrow{B_1: r^{s_1}} \dots q_{n-1} \xrightarrow{B_{n-1}: r^{s_{n-1}}} q \\ \exists i: B_i \cap B \neq \emptyset\}$$

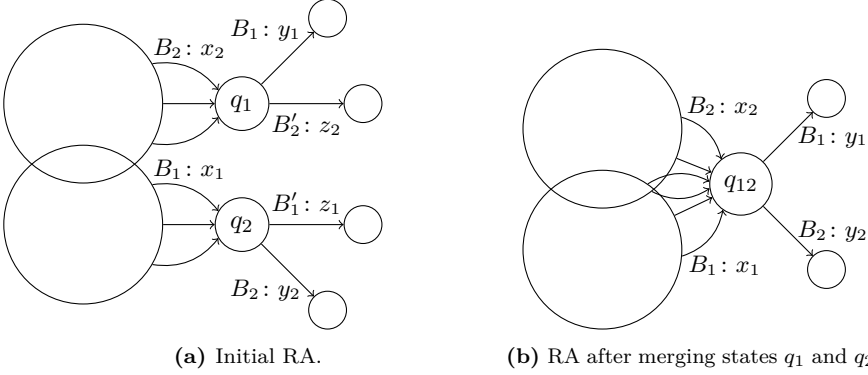


Fig. 5.10. Abstract depiction of merging orthogonal states with $B_1 \cap B'_1 \neq \emptyset \neq B_2 \cap B'_2$. The merged state q_{12} only has the transitions of q_1 and q_2 whose guards are not guaranteed to occur on the respective paths to q_1, q_2 .

Notice that the intersection operation in this definition suffices, since if along every path to a state q at least one basic event $b \in B$ happens, then the event set B as a whole cannot happen after visiting q . In order to compute the set of disabled inputs, we perform a data flow analysis. For this, we apply the worklist algorithm [68], which operates by propagating data flow information along the edges of a graph structure. For each node, the data flow information is combined using a *transfer function*. For computing the disabled inputs, the following transfer functions are employed:

$$\begin{aligned}
 DI(q_0) &:= \emptyset \\
 DI(q) &:= \bigcap_{(p,B) \in \text{pred}(q)} (DI(p) \cup \{B\})
 \end{aligned}$$

with $\text{pred}(q) := \{(p, B) \mid \delta(p, B) = (q, rs), p \neq q\}$ denoting the set of predecessor transitions of a state q . Having set up these preliminary definitions, the concept of orthogonality between states can now be formalized with the following definition:

Definition 5.4 (Orthogonal States). *Let $\mathcal{R}_{\mathcal{T}} = (Q, \delta, q_0)$ be an RA. Let further $p, q \in Q$ be two non-initial distinct states and $B \in \text{BES}(\mathcal{T})$. Then p, q are orthogonal with respect to B iff $B \in DI(p) \cup DI(q)$.*

If upon reaching a state p it is guaranteed that an event in B has occurred, then it is guaranteed that B will not happen starting from state p . Therefore, it does not matter which recovery action will be defined for B in state p since the recovery action will never be applied. Thus, if in state q a specific recovery action has to be performed for B , then we can choose the same recovery action for B in the combined state. Informally, if this holds for all event sets B , then we can think of states p and q as equivalent since their behavior, in terms of recovery actions, is the same, and therefore such states can be merged.

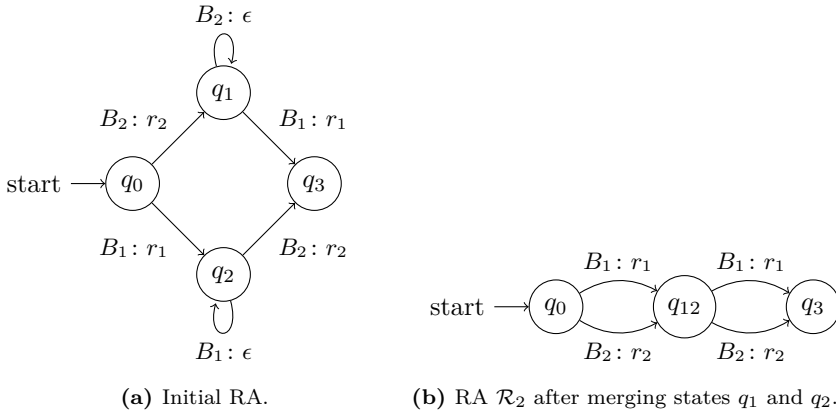


Fig. 5.11. Example merge of orthogonal states.

To illustrate the optimization rule based on the definition of orthogonality, we consider as an example the recovery automaton depicted in Fig. 5.11. This RA reacts to two distinct basic event sets B_1 and B_2 and performs a corresponding recovery action r_1 or r_2 accordingly. An NdDFT that would produce such an RA would be, for example, a system consisting of two parallel spare gates running independently from each other. For the disabled inputs we have:

- $DI(q_0) = \emptyset$,
- $DI(q_1) = DI(q_0) \cup \{B_2\} = \{B_2\}$,
- $DI(q_2) = DI(q_0) \cup \{B_1\} = \{B_1\}$ and
- $DI(q_3) = (DI(q_1) \cup \{B_1\}) \cap (DI(q_2) \cup \{B_2\}) = \{B_1, B_2\}$.

Thus, by Def. 5.4 it holds that q_1 and q_2 are orthogonal with respect to basic event sets B_1 and B_2 . Observe that q_1 has an outgoing loop labeled with $B_2 : \epsilon$ that is disabled. Similarly, q_2 has an outgoing loop labeled by $B_1 : \epsilon$ that cannot occur. In the merged RA, these transitions are eliminated, and all the other incoming and outgoing transitions are redirected to start and end at the merged state, respectively.

We are now ready to incorporate the concept of orthogonality into an equivalence definition for states of recovery automata. For this purpose, we enrich the basic trace equivalence definition from Def. 5.3 as follows:

Definition 5.5 (Syntactical State Recovery Equivalence). *Let $\mathcal{R}_{\mathcal{T}} = (Q, \delta, q_0)$ be an RA. A state-based recovery equivalence $\approx_R \subseteq Q \times Q$ is a maximal relation such that it holds for any states $q_1, q_2 \in Q$ that $q_1 \approx_R q_2$ iff for any $B \in \text{BES}(\mathcal{T})$ it holds that either:*

- $\delta(q_1, B) = (q'_1, rs_1), \delta(q_2, B) = (q'_2, rs_2)$ with $q'_1 \approx_R q'_2, rs_1 = rs_2$ or
- q_1, q_2 are orthogonal with respect to B .

Note that all conditions are syntactical, and we have introduced the necessary means to check them. Next, we aim to formalize the merging procedure for obtaining a new recovery automaton given two recovery-equivalent states. For this we define an *orthogonal merge* operation. Following the process described in the previous example yields the following formal definition, which is visualized by Fig. 5.10 on page 77:

Definition 5.6 (Orthogonal Merge). *Let $\mathcal{R}_{\mathcal{T}} = (Q, \delta, q_0)$ be an RA. The orthogonal merge of recovery-equivalent states $q_1, q_2 \in Q$ is defined by the function $OM(\mathcal{R}, q_1, q_2) := (Q', \delta', q'_0)$ with:*

- $Q' := Q \setminus \{q_1, q_2\} \uplus \{q_{12}\},$
- for all $B \in \text{BES}(\mathcal{T}),$
 - $\delta'(q_{12}, B) := \delta(q_2, B)$ if $B \in \text{DI}(q_1)$ and
 - $\delta'(q_{12}, B) := \delta(q_1, B)$ otherwise,
- $\delta'(p, B) = (q_{12}, rs)$ for any $B \in \text{BES}(\mathcal{T})$ and p such that $\delta(p, B) = (q_1, rs)$ or $\delta(p, B) = (q_2, rs)$ and
- $q'_0 := q_0$ if $q_1 \neq q_0$ and $q_2 \neq q_0,$ and $q'_0 = q_{12}$ otherwise.

Intuitively, the operation replaces the states to be merged by a new combined state, adjusting the transition function accordingly. The initial state is updated if one of the states is also the initial state. Transitions with enabled guards are copied (where the result is guaranteed to be unique). On the other hand, transitions with disabled guards are discarded. We now prove the soundness of the orthogonal merge. The following theorem states under which conditions merging two recovery-equivalent states yields a recovery-equivalent recovery automaton.

Theorem 5.1. *Let $\mathcal{R}_1 = (Q_1, \delta_1, q_{01})$ be an RA and $q_1, q_2 \in Q$ such that $q_1 \approx_R q_2$ and for any $B \notin DI(q_1) \cup DI(q_2)$, it holds for all q'_1, q'_2 such that $\delta(q_1, B) = (q'_1, rs)$ and $\delta(q_2, B) = (q'_2, rs)$, we have $q'_1 \approx_R q'_2$ implies $q'_1 = q'_2$. That is, all equivalent successors have already been merged. Let further $OM(\mathcal{R}_1, q_1, q_2) = \mathcal{R}_2 = (Q_2, \delta_2, q_{02})$ be an RA resulting from the orthogonal merge of q_1 and q_2 . Then $\mathcal{R}_1 \approx_R \mathcal{R}_2$.*

Proof. Let $\beta := B_1, \dots, B_n \in BES(\mathcal{T})^*$ be a sequence of basic event sets produced by an NdDFT. Then $B_i \cap B_j = \emptyset$ for any $i \neq j$. We distinguish two cases:

- Assume \mathcal{R}_1 never visits q_1 or q_2 . By definition of \mathcal{R}_2 we then have that also \mathcal{R}_2 does not visit q_{12} . And by definition of \mathcal{R}_2 again we thus immediately have that $Recovery_{\mathcal{R}_1}(\beta) = Recovery_{\mathcal{R}_2}(\beta)$.
- Assume \mathcal{R}_1 visits q_1 (the case of visiting q_2 is analogous) upon reading B_i for some $i < n$. Now consider B_{i+1} . Let q'_1, q'_{12} and rs_1, rs_{12} be such that:

$$\begin{aligned} \delta_1(q_1, B_{i+1}) &= (q'_1, rs_1) \text{ and} \\ \delta_2(q_{12}, B_{i+1}) &= (q'_{12}, rs_{12}). \end{aligned}$$

By Def. 5.5 this means that we have either:

- q_1, q_2 are orthogonal with respect to B_{i+1} . Then by Def. 5.4 it holds that:

$$B_{i+1} \in DI(q_1) \cup DI(q_2)$$

Assume $B_{i+1} \in DI(q_1)$. Then there exists by construction of DI an index $j < i+1$ such that $B_{i+1} \cap B_j \neq \emptyset$. Contradiction to the definition of β . Hence $B_{i+1} \notin DI(q_1)$. But since $B_{i+1} \in DI(q_1) \cup DI(q_2)$, this implies $B_{i+1} \in DI(q_2)$. Note that $DI(q_2) \subsetneq DI(q_1)$ or otherwise we

obtain again a contradiction to the construction of β . By construction of \mathcal{R}_2 and Def. 5.6 this implies that:

$$(q'_{12}, rs_{12}) = \delta_2(q_{12}, B_{i+1}) = \delta_1(q_1, B_{i+1}) = (q'_1, rs_1)$$

Hence we can conclude $Recovery_{\mathcal{R}_1}(\beta) = Recovery_{\mathcal{R}_2}(\beta)$.

- q_1, q_2 are not orthogonal with respect to B_{i+1} . Then $B_{i+1} \notin DI(q_1) \cup DI(q_2)$ and $rs_1 = rs_{12}$ and $q'_1 \approx_R q'_{12}$. But then by assumption we get $q'_1 = q'_{12}$. We hence obtain $Recovery_{\mathcal{R}_1}(\beta) = Recovery_{\mathcal{R}_2}(\beta)$.

In all cases we have $Recovery_{\mathcal{R}_1}(\beta) = Recovery_{\mathcal{R}_2}(\beta)$ and thus $\mathcal{R}_1 \approx_R \mathcal{R}_2$ by Def. 5.2.

□

5.3.2 Optimizing the FAIL State

The idea of the second case is to identify FAIL states that do not contribute to new recovery action sequences when a set of faults occurs. If a state only leads to a FAIL state, the transition can be turned into a self-loop. Moreover, should the FAIL state no longer be reachable, it can be eliminated. This rule is abstractly illustrated in Fig. 5.12. We further introduce the concept of a FAIL state for recovery automata.

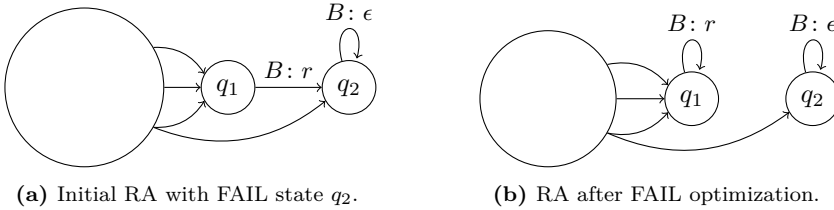


Fig. 5.12. Application of FAIL optimization. The transition $B: r$ can be removed.

Definition 5.7 (FAIL State). *Let $\mathcal{R}_{\mathcal{T}} = (Q, \delta, q_0)$ be an RA and $q \in Q$ a state. Then q is a FAIL state iff for any $B \in BES(\mathcal{T})$, all transitions from q are of the form $\delta(q, B) = (q, \epsilon)$.*

A FAIL state is thus simply a sink state with only ϵ -loop transitions. With the definition of the FAIL state, we can now also formally define an optimization operation similar to the orthogonal merge operation.

Definition 5.8 (FAIL Optimization). Let $\mathcal{R}_{\mathcal{T}} = (Q, \delta, q_0)$ be an RA. The FAIL optimization of a state $q_1 \in Q$ with respect to a FAIL state $q_2 \in Q$ is given by the function $FO(\mathcal{R}, q_1, q_2) := (Q', \delta', q_0)$ with:

- $Q' := Q \setminus \{q_2 \mid \neg \exists p \in Q \setminus \{q_2\}, B \in BES(\mathcal{T}) : \delta(p, B) = (q_2, rs)\}$ and
- $\delta'(q_1, B) := (q_1, rs)$ if $\delta(q_1, B) = (q_2, rs)$ for all $B \in BES(\mathcal{T})$.

Note that the FAIL optimization does not apply to all pairs of states and FAIL states. We show this property in the following lemma. Intuitively, the FAIL rule cannot be applied when a state has multiple non- ϵ transitions.

Lemma 5.1. *There exists a recovery automaton $\mathcal{R} = (Q, \delta, q_0)$ with state $p \in Q$ and FAIL state $q \in Q$ such that \mathcal{R} is not recovery-equivalent to $FO(\mathcal{R}, p, q)$.*

Proof. Consider the recovery automaton $\mathcal{R} := (Q, \delta, q_0)$ and states q_0, q_1 as given in Fig. 5.13a. We show that \mathcal{R} and the states q_0, q_1 fulfill the conditions. Let further $\mathcal{R}' := FO(\mathcal{R}, q_0, q_1)$ be the recovery automaton obtained through the FAIL optimization of q_0 with respect to the FAIL state q_1 , as depicted in Fig. 5.13b. For the input sequence $\beta := B_1 B_2$ we now have:

$$\text{Recovery}_{\mathcal{R}}(\beta) = r_1 \neq r_1 r_2 = \text{Recovery}_{\mathcal{R}'}(\beta)$$

By Def. 5.5 we thus obtain that \mathcal{R} and \mathcal{R}' are not recovery-equivalent.

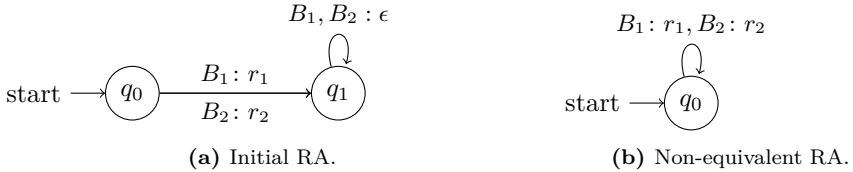


Fig. 5.13. RA where the FAIL rule is not applicable. In the original automaton, only one of the event sets B_1 or B_2 can trigger a recovery action. The FAIL optimized RA, however, allows for both B_1 and B_2 to trigger a recovery action.

□

Next, we show soundness of the FAIL optimization. The following theorem can capture the conditions for soundness of the formalized optimization operation:

Theorem 5.2. *Let $\mathcal{R}_1 = (Q_1, \delta_1, q_{01})$ be an RA with a pair of states q_1 and q_2 such that q_2 is a FAIL state and all transitions of q_1 are ϵ -loops except for one*

5.3. FURTHER OPTIMIZATION OF RECOVERY AUTOMATA

transition being of the form $\delta_1(q_1, B) = (q_2, rs)$, such that $rs \neq \epsilon$. Let further $FO(\mathcal{R}, q_1, q_2) = \mathcal{R}_2 = (Q_2, \delta_2, q_{02})$ be the FAIL-optimized RA. Then $\mathcal{R}_1 \approx_R \mathcal{R}_2$.

Proof. Let $\beta := B_1, \dots, B_n \in BES(\mathcal{T})^*$ be a sequence of basic event sets produced by an NdDFT. Then $B_i \cap B_j = \emptyset$ for any $i \neq j$. We distinguish two cases:

- Assume \mathcal{R}_1 never visits q_1 . Then by definition of \mathcal{R}_2 , it also never visits q_1 . As both automata are defined to be equal otherwise, we then immediately have that $Recovery_{\mathcal{R}_1}(\beta) = Recovery_{\mathcal{R}_2}(\beta)$.
- Assume \mathcal{R}_1 visits q_1 upon reading B_i for some $i < n$. Then by definition, \mathcal{R}_2 also visits q_1 upon reading B_i . Now consider B_{i+1} . By the construction of \mathcal{R}_2 , it holds that $\delta_1(q_1, B_{i+1}) = (q_2, rs)$ and $\delta_2(q_1, B_{i+1}) = (q_1, rs)$ for some recovery action sequence rs (I). Since q_2 is a FAIL state, we obtain from Def. 5.7 that $\delta_1(q_2, B_j) = (q_2, \epsilon)$ for any $j > i + 1$ (II). Moreover, since also $B_j \cap B_{i+1} = \emptyset$ for any $j > i + 1$ we also have by definition of q_1 and \mathcal{R}_2 that $\delta_2(q_1, B_j) = (q_1, \epsilon)$ (III). In total, we can therefore conclude that:

$$\begin{aligned}
 Recovery_{\mathcal{R}_1}(\beta) &= Recovery_{\mathcal{R}_1}(B_1, \dots, B_n) && \text{(Def. } \beta) \\
 &= Recovery_{\mathcal{R}_1}(B_1, \dots, B_i, B_{i+1}) && \text{(II)} \\
 &= Recovery_{\mathcal{R}_2}(B_1, \dots, B_i, B_{i+1}) && \text{(I)} \\
 &= Recovery_{\mathcal{R}_2}(B_1, \dots, B_n) && \text{(III)} \\
 &= Recovery_{\mathcal{R}_2}(\beta) && \text{(Def. } \beta)
 \end{aligned}$$

In all cases $Recovery_{\mathcal{R}_1}(\beta) = Recovery_{\mathcal{R}_2}(\beta)$. Hence, $\mathcal{R}_1 \approx_R \mathcal{R}_2$ by Def. 5.2. \square

To conclude on the FAIL rule, we prove that the orthogonal merge rule does not cover the FAIL optimization rule. This is done by giving a concrete example where the orthogonal merge cannot remove any states, yet a smaller recovery automaton can be obtained by applying the FAIL rule.

Lemma 5.2. *There exists a recovery automaton $\mathcal{R} = (Q, \delta, q_0)$ such that:*

- for any states $p, q \in Q$ with $p \neq q$ it holds that $p \not\approx_R q$ and
- there exists a recovery-equivalent recovery automaton with fewer states that can be obtained through the FAIL optimization rule.

Proof. Consider the following recovery automaton $\mathcal{R} := (Q, \delta, q_0)$ as illustrated in Fig. 5.14a. We show in the following that \mathcal{R} fulfills the desired conditions. First of all, it holds that:

$$B_3, B_4 \notin DI(q_1) \cup DI(q_2) = \{B_1\} \cup \{B_2\}$$

Furthermore, for any states $p, q \in Q$ with $p \neq q$, there exists a B such that $\delta(q, B) = (r, q'), \delta(p, B) = (r', p')$ with $r \neq r'$. Taking these two together, we directly obtain $p \not\approx_R q$ for any states $p \neq q \in Q$ by Def. 5.5. On the other hand, q_3 is a FAIL state. Also, both pairs q_1, q_3 and q_2, q_3 fulfill the conditions for sound application of the FAIL rule according to Theorem 5.2, as they possess exactly one non- ϵ transition leading into a FAIL state. The reduced RA $\mathcal{R}' := FO(FO(\mathcal{R}, q_1, q_3), q_2, q_3)$ with the FAIL state eliminated is illustrated in Fig. 5.14b.

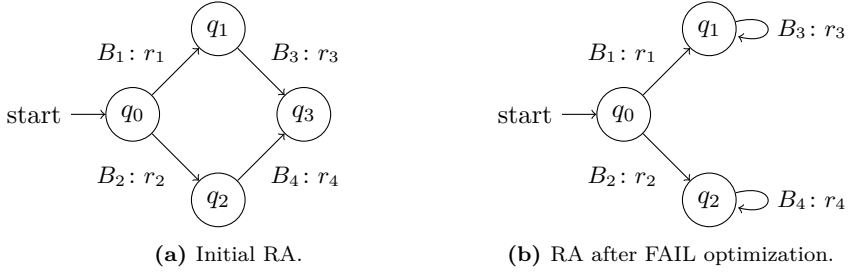


Fig. 5.14. RA that requires FAIL rule for optimization. The orthogonal merge is not applicable to the original automaton. The FAIL optimization, however, can be performed to obtain a smaller automaton.

□

5.3.3 Completeness

Having established two optimization rules, the question arises if these rules are now complete. That is, if we consider an arbitrary recovery automaton, do the optimization rules yield a minimal recovery automaton at all times? This question is investigated in this section. The investigation is carried out roughly according to the following steps: First, we define the recovery equivalence relation on the state level to allow for comparison with the state-based syntactical recovery equivalence. Using this, we show that the optimization rules we provided are indeed generally not sufficient to characterize recovery equivalence. In

the following, we then define a class of recovery automata that excludes the problematic instances and for which we can thus prove the completeness of our approach.

Definition 5.9 (Semantic State Recovery Equivalence). *Semantic state-based recovery equivalence $\sim_R \subseteq Q \times Q$ is a maximal relation such that it holds for any states $q_1, q_2 \in Q$ that $q_1 \sim_R q_2$ iff $\delta^*(q_1, w) = \delta^*(q_2, w)$ for any $w = B_1, \dots, B_n$ with:*

- $B_i \cap B_j = \emptyset$ for any $i \neq j$ and
- $B_i \notin DI(q_1) \cup DI(q_2)$ for all i .

The definitions of state-based and semantic recovery equivalence are nearly identical, except that the latter disregards basic event sets that are excluded by disabled inputs. This additional property ensures that we can only continue from a given state using basic event sets that are not disabled due to having already occurred. The connection between both concepts is given through the recovery equivalence of the initial states. The following lemma captures this property.

Lemma 5.3. *Let $\mathcal{R}_1 = (Q_1, \delta_1, q_{01})$ and $\mathcal{R}_2 = (Q_2, \delta_2, q_{02})$ then $\mathcal{R}_1 \approx_R \mathcal{R}_2$ iff $q_{01} \sim_R q_{02}$.*

Proof. We have $DI(q_{01}) = DI(q_{02}) = \emptyset$ due to q_{01}, q_{02} being initial states. Furthermore, it holds that:

- $\delta^*(q_{01}, B_1 \dots B_n) = \text{Recovery}_{\mathcal{R}_1}(B_1, \dots, B_n)$ and
- $\delta^*(q_{02}, B_1 \dots B_n) = \text{Recovery}_{\mathcal{R}_2}(B_1, \dots, B_n)$.

for any sequence B_1, \dots, B_n by definition of δ^* . Hence we obtain the equivalence by Def. 5.9 and Def. 5.2. \square

5.3.3.1 General Incompleteness

We now show that our approach is generally not optimal. This is achieved by constructing a recovery automaton that cannot be optimized further by any of the provided optimization rules but still contains recovery-equivalent states. We also show that by exploiting these recovery-equivalent states, it is indeed possible to obtain a smaller recovery-equivalent recovery automaton.

Lemma 5.4. *There exists a recovery automaton \mathcal{R} such that:*

- $\sim_R \not\subseteq \approx_R$,
- the FAIL rule cannot be applied and
- there exists a recovery-equivalent recovery automaton with fewer states.

Proof. Consider the recovery automaton $\mathcal{R} := (Q, \delta, q_0)$ as illustrated in Fig. 5.15a on the following page. We show in the following that \mathcal{R} fulfills all listed conditions. Consider further the states q_1 and q_2 . We have the following two properties:

- Both q_1 and q_2 yield the recovery action r_0 upon reading B and ϵ for any other input.
- In the successors p_1 and p_2 , the input B can no longer be read if reached via q_1 or q_2 , respectively.

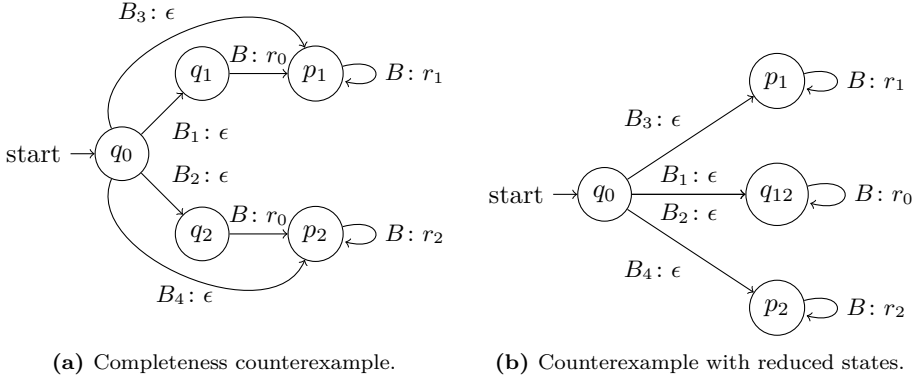
Hence $q_1 \sim_R q_2$ by Def. 5.9. Furthermore, since p_1 and p_2 yield $r_1 \neq r_2$ respectively upon reading B , it also holds that $p_1 \not\sim_R p_2$. We also have that $DI(p_1) = \emptyset$ and $DI(p_2) = \emptyset$ and thus in particular $B \notin DI(p_1) \cup DI(p_2)$. Hence p_1 and p_2 are not orthogonal with respect to B . In total we also obtain $p_1 \not\approx_R p_2$ by Def. 5.4. Furthermore, we also have:

$$B \notin DI(q_1) \cup DI(q_2) = \{B_1\} \cup \{B_2\}$$

Thus again by Def. 5.4, we finally obtain $q_1 \not\approx_R q_2$, which implies $\sim_R \not\subseteq \approx_R$. As for the FAIL rule, clearly there exists no FAIL state, therefore, it cannot be applied. Fig. 5.15 on the following page shows the optimized counterexample RA with fewer states. □

5.3.3.2 Restricted Completeness

The key issue leading to incompleteness using a merging-based approach lies in the property that it is generally possible for two states to be recovery-equivalent but to have non-recovery-equivalent successors. This property was also present in the counterexample of Lemma 5.4. However, requiring equivalent successors is an intrinsic part of the definition of \approx_R . The only exceptions are successors that cannot be reached due to the respective guards being in the set of disabled inputs.



(a) Completeness counterexample.

(b) Counterexample with reduced states.

Fig. 5.15. Completeness counterexample that cannot be obtained with optimization rules. Neither the orthogonal merge, nor the FAIL optimization can be applied to the original automaton. However, a smaller recovery equivalent automaton exists.

Definition 5.10. (*Successor Consistency*) An RA is consistent with respect to \sim_R iff for any p, q with $p \sim_R q$ it holds for any $B \notin DI(p) \cup DI(q)$ and p', q' successors of p, q that $p' \sim_R q'$. The class of RAs consistent with respect to \sim_R is defined as

$$\mathcal{C} := \{\mathcal{R} \mid \mathcal{R} \text{ is consistent with respect to } \sim_R\}$$

Under which conditions the synthesized recovery automata are guaranteed to be in this class is not clear and a problem for future investigation. In addition to being a member of the class \mathcal{C} , to simplify the proofs, we also introduce a normal form for recovery automata in the following. This normal form will guarantee later on that it is possible to reach each state backwards from a sink state.

Definition 5.11 (Normal form). A recovery automaton \mathcal{R} is in normal form iff there exist no cycles, except for loop transitions.

A recovery automaton can be easily transformed into normal form. The idea is to exploit again the property that every basic event set can occur at most once.

Lemma 5.5. Each RA \mathcal{R} can be transformed into a recovery-equivalent RA in normal form.

Proof. Since every input can occur at most once, an equivalent RA can be gained by loop unrolling and by removing disabled transitions. \square

However, unrolling an RA may lead to the introduction of new states. Since the goal is the reduction of states, this is undesirable. In the special case of recovery automata synthesized from an FT, according to the given construction, however, no additional states need to be introduced. This is due to the fact that these recovery automata are already in normal form.

Lemma 5.6. *If \mathcal{R} is an RA synthesized from an FT, then \mathcal{R} is already in normal form.*

Proof. Since faults are permanent, the given state space generation for transforming an NdDFT to a MA creates successor states with strictly monotonically increasing failed events. Hence, the algorithm does not add any cycles. The extraction process of an RA from an MA chiefly involves removing nodes and edges and only introduces new edges of the form $\delta(q, B) = (q, \epsilon)$, i.e, self-loops. Therefore, the RA also remains free of cycles except for self-loops. \square

As stated earlier, the primary purpose of the normal form is to ensure that any state can be reached backwards from a sink state. This property is captured in the following lemma.

Lemma 5.7. *Let \mathcal{R} be an RA be in normal form, then for any state q , there exists a path q, \dots, q_{sink} such that q_{sink} is a sink state.*

Proof. Since the state space of \mathcal{R} is finite, and there exist no cycles except for self-loops, every state must have a path to a sink state. \square

The proof strategy is now as follows: In order to show completeness on the restricted class \mathcal{C} , we first provide a fixpoint characterization of the semantic state-based recovery equivalence. We show the correctness of the characterization by proving that the fixpoint is indeed the semantic state-based recovery equivalence. Using this characterization, we can then show the main claim by showing that the syntactical state-based recovery equivalence captures each iteration.

Definition 5.12 (Fixpoint Characterization). *Define the fixpoint iteration $F : \mathbb{N} \rightarrow (Q \times Q)$ with*

$$\begin{aligned}
 F_0 &:= \{(q, q) \mid q \in Q\} \cup \{(p, q) \mid p, q \in Q, p, q \text{ sink states with } p \sim_R q\} \\
 F_n &:= F_{n-1} \cup \{(q_1, q_2) \mid q_1 \sim_R q_2, \forall B \notin DI(q_1) \cup DI(q_2) : \\
 &\quad \exists q'_1, q'_2 : \delta(q_1, B) = (q'_1, rs), \delta(q_2, B) = (q'_2, rs), \\
 &\quad (q'_1, q'_2) \in F_{n-1} \text{ or } q'_1 = q_1 \text{ and } q'_2 = q_2\}
 \end{aligned}$$

Let $\text{Fix}(F) := F_n$ denote the fixpoint of F such that $F_n = F_{n+1}$.

Note that since F is monotonic by definition, and the state space Q is finite, $\text{Fix}(F)$ is guaranteed to exist. Intuitively, the iteration function F initially contains all pairs of directly equivalent states, either because they are equal or because they have no successors. That is, they are sink states. In each following iteration, pairs of states are then added so that all reachable successors of these pairs are already equivalent. The notion of reachability here also considers orthogonality, only considering transitions that are not disabled due to their guards being in the set of disabled inputs. If no further pairs of states can be added, then the fixpoint $\text{Fix}(F)$ is reached. We prove with the following lemma for the class of recovery automata consistent with respect to \sim_R , that are also in normal form, that the fixpoint characterization indeed captures the recovery equivalence \sim_R .

Lemma 5.8. *If \mathcal{R} is an RA in normal form and $\mathcal{R} \in \mathcal{C}$, then $\text{Fix}(F) = \sim_R$.*

Proof. It holds that $F_n \subseteq \sim_R$ for any n by definition of F_n . We now show that $\sim_R \subseteq F_n$ for some n . Let q_n denote a state such that any path of length at least n ends in a sink state. The existence of the sink state is guaranteed by Lemma 5.7 since \mathcal{R} is in normal form by assumption. Proof by induction over n .

- Consider $n = 0$. Then q_0 and p_0 are sink states. If $q_0 \sim_R p_0$, then in total $(q_0, p_0) \in F_0$ by definition.
- Now assume if $p_i \sim_R p_j$ then $(p_i, q_j) \in F_k$ for some k for any $i, j < n$ (I). Consider some $p_n \sim_R q_m$ with $m \leq n$. Let $B \notin DI(p_n) \cup DI(q_m)$. Consider the following cases:
 - $\delta(q_m, B) = (q_m, rs)$ and $\delta(p_n, B) = (p_n, rs)$. Then by Def. 5.12, $(q_m, p_n) \in F_j$ is not required for any j .
 - $\delta(q_m, B) = (q_{m-1}, rs)$ and $\delta(p_n, B) = (p_{n-1}, rs)$. Since $\mathcal{R} \in \mathcal{C}$, it holds by Def. 5.10 that $p_{n-1} \sim_R q_{m-1}$. Hence by induction hypothesis (I) $(p_{n-1}, q_{m-1}) \in F_j$ for some j .

In the following, we consider the cases where there is a single state change. Proof by induction over m .

- Consider $m = 0$. Then $\delta(q_m, B) = (q_m, rs)$ and $\delta(p_n, B) = (p_{n-1}, rs)$. Since $\mathcal{R} \in \mathcal{C}$ it holds by Def. 5.10 that $p_{n-1} \sim_R q_m$. Hence by induction hypothesis (I) $(p_{n-1}, q_m) \in F_j$ for some j .

- Consider $m > 0$. Assume if $p_n \sim_R p_i$ then $(p_n, q_i) \in F_j$ for some j for any $i < m$ (II). Assume without loss of generality that the state change occurs in q_m , otherwise swap p and q . Then $\delta(q_m, B) = (q_{m-1}, rs)$ and $\delta(p_n, B) = (p_n, rs)$. Since $\mathcal{R} \in \mathcal{C}$, it holds by Def. 5.10 that $p_n \sim_R q_{m-1}$. Hence, by induction hypothesis (II) $(p_n, q_{m-1}) \in F_j$ for some j .

In all cases, this yields in total $(p_n, q_m) \in F_{j+1}$ for some j by Def. 5.12.

By induction principle this yields in total that if $p \sim_R q$ then $(p, q) \in F_n$ for some n . Since further by definition $F_n \subseteq F_{n+1}$ for any n , we hence obtain in total that there exists some n such that for any $p \sim_R q$ we have $(p, q) \in F_n$. \square

Using the fixpoint characterization, it is now easier to show what kind of states are recovery-equivalent by considering the individual fixpoint iterations and the pairs of equivalent states added in them, respectively. With this, it is now possible to move toward the main equivalence theorem. To this aim, we first show that generally, any recovery-equivalent pair of states discovered by the fixpoint iteration is also discovered by the syntactical recovery equivalence.

Lemma 5.9. *For any n it holds that $F_n \subseteq \approx_R$.*

Proof. Proof by induction over n .

- For $n = 0$ we have $F_0 \subseteq \approx_R$.
- Assume the hypothesis holds for some n . Let $(q_1, q_2) \in F_{n+1}$. If $(q_1, q_2) \in F_n$ then by induction hypothesis also $q_1 \approx_R q_2$. Now consider the case $(q_1, q_2) \notin F_n$. Since $(q_1, q_2) \in F_{n+1}$ we have $q_1 \sim_R q_2$ (I). Assume $q_1 \not\approx_R q_2$. Then there exists B with $\delta(q_1, B) = (q'_1, rs_1)$, $\delta(q_2, B) = (q'_2, rs_2)$, q_1, q_2 not orthogonal with respect to B and $rs_1 \neq rs_2$ or $q'_1 \not\approx_R q'_2$.
 - Assume $rs_1 \neq rs_2$. Then $\delta^*(q_1, B) \neq \delta^*(q_2, B)$. Since q_1, q_2 are not orthogonal with respect to B , it holds that $B \notin DI(q_1) \cup DI(q_2)$. In total, this gives $q_1 \not\approx_R q_2$. Contradiction to (I).
 - Assume $rs_1 = rs_2$. Then $q'_1 \not\approx_R q'_2$. By definition of F_n , we also have $(q'_1, q'_2) \in F_n$. Hence by induction hypothesis, it also holds that $q'_1 \approx_R q'_2$. Contradiction.

In total, we thus obtain $q_1 \approx_R q_2$.

\square

By combining the individual preliminary results, the desired main result now follows.

Theorem 5.3. *If \mathcal{R} is an RA in normal form and $\mathcal{R} \in \mathcal{C}$, then $\sim_R \subseteq \approx_R$.*

Proof. Since \mathcal{R} is in normal form and $\mathcal{R} \in \mathcal{C}$ it holds that:

$$\begin{aligned} \sim_R &= \text{Fix}(F) && \text{(Lemma 5.8)} \\ &\subseteq \approx_R && \text{(Lemma 5.9)} \end{aligned}$$

□

5.4 Modular Synthesis of Recovery Automata

The Markovian state space generated from a fault tree can be massive. In general, the corresponding state space size can grow exponentially with the number of nodes in a fault tree. The problem of an exponentially increasing state space is commonly known as the *state space explosion problem*. In conventional dynamic fault trees, the blow-up can be mainly attributed to the interleaving occurrence of basic events. In the case of non-deterministic DFTs, the state space explosion problem gains an additional dimension: The non-determinism caused by selecting an appropriate recovery action generates an additional source of exponential blow-up. Finally, when considering repair events, both exponential blow-up sources become even worse. As failed events can now be repaired, additional interleavings in the occurrence of basic events become possible. Likewise, repaired spares or primaries increase the number of selectable recovery actions generated by SPARE gates. The presence of repair actions also disables useful techniques for state space reduction. Consider, for example, a simple static fault tree consisting of just one OR gate with n basic events as inputs. In the non-repairable case, the Markovian state space can be represented with just two states, as every occurrence of a basic event leads directly into the FAIL state. In the repairable case, however, the state must concretely remember the failed basic event, leading to the necessity of introducing n new states.

The varying sources of and issues amplifying the state space explosion problem make it abundantly clear: Ensuring scalability while synthesizing recovery strategies for large fault trees with hundreds of basic events is nearly impossible using the previous, naive workflow. In the following, we consider how existing modular approaches for deterministic DFTs can be leveraged to solve the synthesis problem.

5.4.1 Modular Workflow

Previous works have considered employing modularization techniques to tackle the state space explosion problem for calculating RAMS metrics on deterministic DFTs. These primarily involve detecting independent sub-trees in a fault tree – referred to as modules –, evaluating the metrics on the individual modules, and then composing them into the total metric for the original fault tree. For example, the total reliability after a certain time span for two modules connected via an AND gate can be obtained through multiplication.

Commonly, this approach faces a significant issue: Not all metrics can be computed compositionally but instead require the full state space for computation. In particular, essential metrics such as the MTTF are not compositional [62]. However, in the context of the recovery strategy synthesis problem, the problem of compositionality changes. Even though the metric to be optimized may not be compositional, for computing the recovery automaton, it is fortunately not necessary to compose the actual metric for the complete tree. Instead, the objects that require composition are the already optimized recovery automata. Automata composition in turn is a common problem that can be solved using standard techniques. We therefore exploit a two-stage approach by first synthesizing recovery automata in a modular fashion and then employing them during the computation of the actual metrics. In this manner, the non-determinism can be resolved modularly during the synthesis step. In greater detail, we apply the following approach:

1. Modularization: determine the modules in the fault tree.
2. Trimming: discard modules without non-determinism.
3. Synthesis: compute the optimal RA for each module, and reduce it.
4. Composition: assemble the overall recovery automaton from the modular RA.

As noted previously, basic events are a significant driver for exponential blow-up. Therefore, events that do not affect the resolution of the non-determinism are taken out of the equation. Finally, as the non-determinism has already been resolved before the evaluation step, this particular source of exponential blow-up is absent during the computation of the metrics. The new workflow incorporating modularization is visualized in Fig. 5.16 on the following page.

Note that there is no additional RA reduction step after the composition as the experiments in Sec. 8.3 on page 154 revealed that a final reduction does not yield any notable improvements.

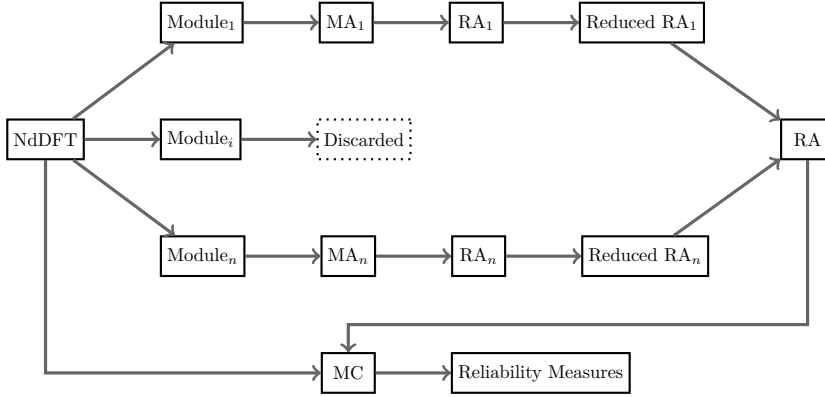


Fig. 5.16. Transformation road map with modularization. Modules without SPARE gates are discarded.

5.4.2 Modularization

We base our modularization approach on the pre-existing algorithm given in [69]. It applies a depth-first search on the fault tree, traversing all nodes while keeping track of each node's first and last visiting time. These visiting times are then used to identify the modules using the following criterion: Given a node that is suspected of being the root of a module, if its descendants' visit dates - both first and last - all lie within the first and last visit dates of that node, then the node and all of its descendants form a module. In addition to this basic rule, further restrictions have to be applied to obtain the desired compositionality property for the recovery automata.

Two special cases have to be considered: SPARE gates and all types of priority gates. Priority gates are road blockers to the desired compositionality property, as they may change the optimization direction. Consider, for example, a POR gate. In the case of the first input being a SPARE gate, the optimal strategy for maximizing the MTTF would also be to maximize the MTTF of the SPARE gate. In other words, claiming its available spares is the best course of action. On the other hand, if the SPARE gate were the second input to a POR gate, then suddenly this simple relationship changes: Now minimizing the MTTF of the SPARE gate will lead to a scenario where the POR gate is more inclined to become fail-safe. The two scenarios are visualized in Fig. 5.17 on the next page.

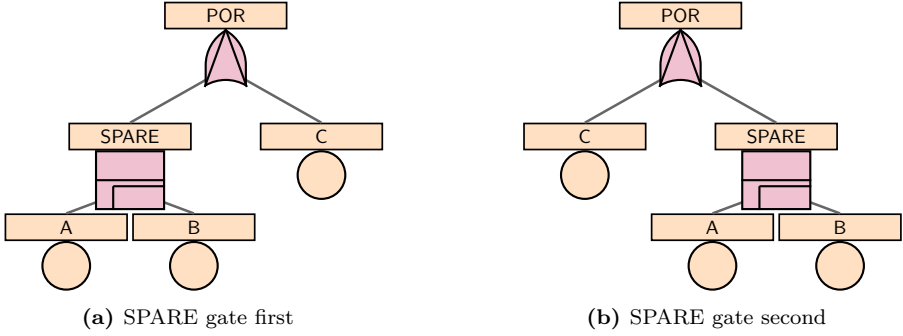


Fig. 5.17. Non-compositionality of priority gates as they change optimization direction.

Therefore, given recovery automata for two modules connected by a POR node, we cannot obtain the overall recovery automaton utilizing composition. In addition to priority gates, SPARE gates also prohibit further modularization of their sub-trees. Due to the semantic definition of a SPARE gate, any basic event contained in a sub-tree may trigger a recovery action and thus requires a representation within the Markovian state space. Bundling these observations, we obtain the following restrictions of the modularization rules:

- A SPARE gate that is a descendant of a priority gate cannot be the module's root.
- A node that has a SPARE gate as a descendant and that is a descendant of a priority gate cannot be the root of a module.
- A descendant of a SPARE gate cannot be the root of a module.

Finally, an example application of the algorithm with the additional rules is given in Fig. 5.18 on the following page. The algorithm proceeds in a leftmost order. Each node is labeled by the first and last visiting time, and dotted boxes indicate the computed modules. All modules without SPARE gates can be trimmed, leaving only the bottom-right module for the actual synthesis.

5.4. MODULAR SYNTHESIS OF RECOVERY AUTOMATA

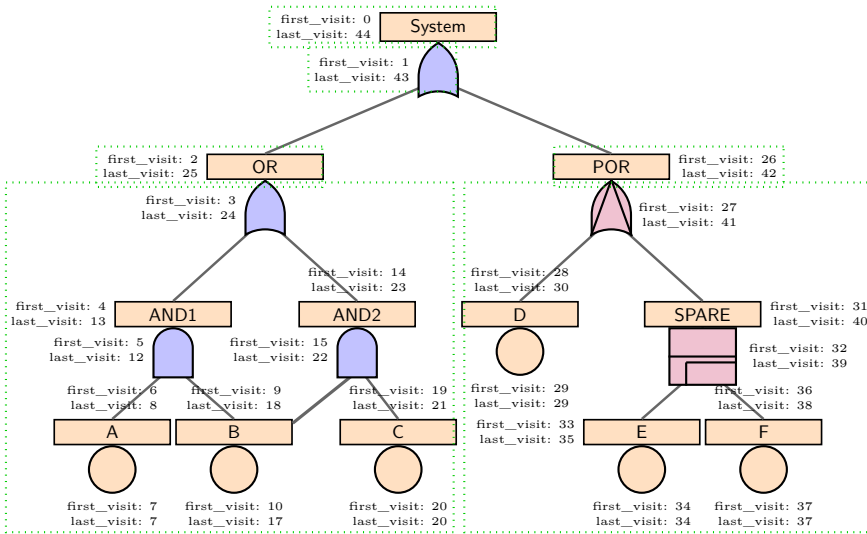


Fig. 5.18. Example application of the modularization algorithm. All modules except for the bottom-right module can be discarded.

Chapter 6

Partial Observability

All events in a fault tree are fully observable. All previous considerations have been considered under this premise. In reality, however, faults are not always fully observable at all times, or in other words, these faults are only *partially observable*. In this chapter, we consider what limitations and challenges partial observability impose on DFTs and investigate how we can lift the NdDFT approach to deal with them. Partial observability can take on several forms. Some examples include:

- Faults might be noticed with some time delay. For example, a watchdog that checks for faulty behavior at some defined time interval can only report a fault during those timed checks. Until the watchdog reveals the fault, it can freely propagate in the system.
- Faults might only be noticed at a higher system level. For example, equipment malfunctions, but this malfunction is not observed directly due to a lack of monitoring. Only the propagated fault is then indirectly observed through the malfunction of a higher-level system function.
- Monitoring processes or equipment can fail. A fault might not be observed because the monitor responsible for reporting on the fault suffered a critical failure and is not working.

When viewed together with SPARE gates, these effects have dire consequences: If a fault is not observed - or alternatively observed with a time delay - then the

FDIR must perform recovery with only the given limited knowledge. Consider, for example, a system with two SPARE gates, which share a common spare. However, only the system failure can be observed. From this information, how should the FDIR proceed in order to recover?

This question leads back to the previous approach: Non-determinism. The following considers a further extension that incorporates observability constraints into the decision-making of the recovery automaton. Several challenges need to be addressed: First of all, to model the observability constraints, a new fault tree element is needed. It will mark which events can be observed and the expected observation delay. Furthermore, since it can now be unknown whether an event has occurred but not been observed, the Markovian state space construction needs to be adapted. The states need to represent this uncertainty in knowledge. The recovery automaton model also requires adjustments to its input language. Previously it was defined to consume inputs of basic event sets. Now it has to react to observations, which might again occur simultaneously. Also, since observation delays might occur, the recovery automaton also needs to be extended with some notion of timed behavior. Finally, topics such as modularization have to be revisited in order to lift the observable to the partially observable model.

6.1 Partially Observable Dynamic Fault Trees

We extend the NdDFTs to a more expressive model we call Partially Observable Dynamic Fault Tree (PODFT). This model introduces a new gate type, a so-called MONITOR gate, that declares which events in the fault tree are observable. Hence, also which events can be reacted to by means of a recovery action. In the case of all events being observable, we obtain a classical NdDFT. [Fig. 6.1 on the following page](#) gives an overview of the different relevant DFT classes introduced up until now.

The regular classes of DFT and Repairable DFT known from existing literature are depicted at the bottom of the hierarchy. Traversing upwards, new features such as non-determinism and partial observability are added. The new classes discussed in the chapter will be the PO classes. In the following, we define the necessary syntactic and semantic extensions to events, gates, and the propagation logic.

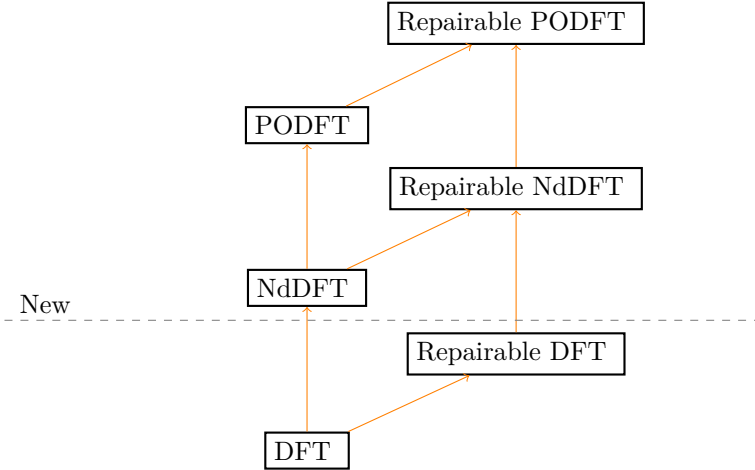


Fig. 6.1. Class hierarchy of DFT classes.

6.1.1 MONITOR Gate

Syntactically, a PODFT is a DFT with an additional gate: A MONITOR gate. The MONITOR gate augments the PODFT with the information by which events can be observed. In order to incorporate time delays for an observation, we parameterize the MONITOR gate by a time parameter t . Such a time delay could, for example, model a watchdog that only performs a periodic observation every t seconds. Hence, we obtain the t -MONITOR gate for which we define the following basic properties. A t -MONITOR gate has:

- any number of observation inputs,
- an optional fault input for failing,
- a fault output for propagating failure, and
- an optional observation delay time t .

The gate symbol with its in- and outputs is given in Fig. 6.2 on the next page. The primary input at the center is the observation input, and the input to the small box on the bottom left is the fault input. If the fault input is triggered, the MONITOR gate is set to fail and can no longer perform observations. Furthermore, its failure is propagated using its fault output. This

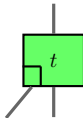


Fig. 6.2. t -MONITOR gate. The right-side input is the observation input. The left-side input causes the gate to fail. The parent propagates the gate failure.

allows modeling constructs such as redundant observers. If the fault input is repaired, observations are re-enabled. Multiple fault inputs can be achieved by pre-positioning an OR gate.

For a given MONITOR gate m we use the shorthand notation $t(m)$ to denote its observation delay t . We also call 0-MONITOR gates immediate MONITOR gates and t -MONITOR gates with $t > 0$ delayed MONITOR gates. The t -MONITOR gate integrated into the DFT definition yields the syntactic definition for PODFTs.

Definition 6.1. (*Partially Observable Dynamic Fault Tree*) A *Partially Observable Dynamic Fault Tree (PODFT)* \mathcal{T} is a FT with:

$$\text{PODFTGates} = \text{DFTGates} \cup \{t\text{-MONITOR} \mid t \in \mathbb{R}, t \geq 0\}$$

6.1.2 Gate and Event Semantics

In the following, we discuss the semantics of the observation process and its consequences on the semantics of other gates and events. First of all, for the repair of basic events, we require a more flexible model than the old static rate model. In NdDFTs, transient and repairable faults behaved semantically the same.

However, this equivalence no longer upholds in PODFTs. For transient failures, repair may start even when it is not known if a BE occurred. For example, a transient bitflip may appear and disappear without an observational effect. Repairs that model an active action, on the other hand, should in many cases only be started upon observing some fault. We therefore extend the repair rate model of BEs to allow a change in the repair rate based on the current observations. Formally, the repair rate association $R(b)$ for a basic event b changes to $R(b, O)$ where O is a set of observed events.

Possible timelines exhibiting fault occurrence, observation, and repair are shown in Fig. 6.3 on the following page. For simplicity, we leave out the

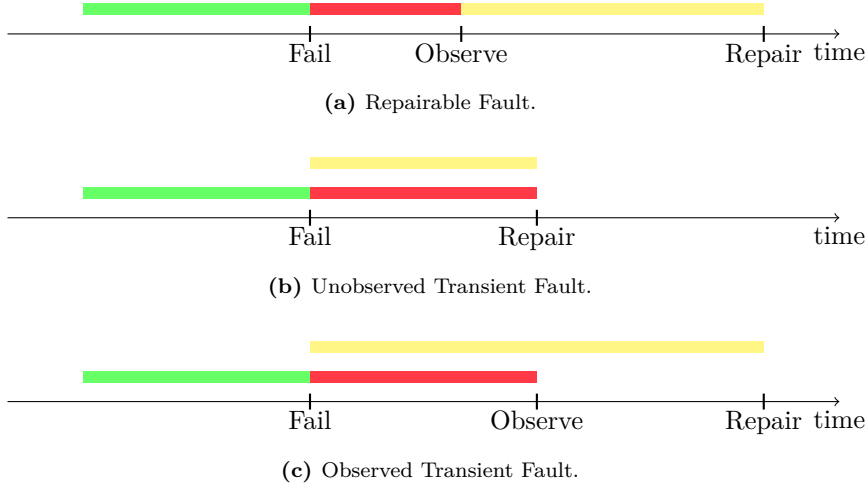


Fig. 6.3. Timelines for repairable and transient faults. Green represents being in an operational state, red in a failed state and yellow an initiated repair process.

observation of the repair event. In the classical case of Fig. 6.3a, where a fault is repaired through a repair process, say a reset, we only start the reset upon observation of the fault. Transient faults may self-repair without requiring an observation. In fact, a transient fault might even be repaired without being observed at all, as illustrated in Fig. 6.3b. Finally, Fig. 6.3c shows the case of a transient fault being observed before it vanishes.

To adapt the gate semantics, in addition to adapting the fault propagation, we also need to consider the propagation of observation information. Fig. 6.4 on the next page gives an example where the TLE is not observable, yet its observation state can be fully deduced. In this case, all basic events are observable, effectively reducing the PODFT to a regular DFT. When Unit_1 and Unit_2 are observed, we can infer the failure of the AND gate. Thus, we can also infer the failure of the System SPARE gate and then perform a switch to Spare as a reaction. Generally, we extend the propagation semantics of each gate to propagate observation according to the same rules for propagating failure. E.g., AND gates propagate if the failure of all inputs has been observed. Likewise, OR gates propagate if the failure of any input has been observed. Further illustrations of the PODFT syntax, gate interactions, and semantic interpretations can be found together with the synthesis examples in Section 6.5.

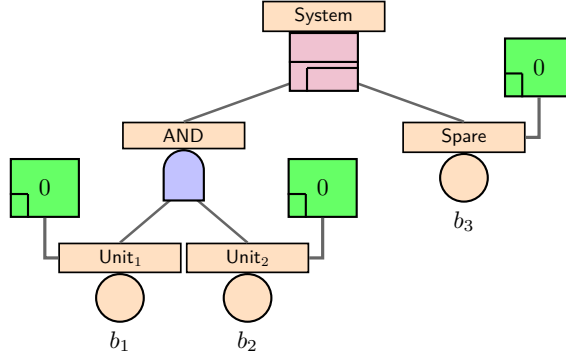


Fig. 6.4. Indirectly fully observable PODFT. The AND and System events are not directly observable, but their occurrence can be inferred from the other observations.

6.2 Belief Markov Automaton Semantics

This section focuses on the two-stage approach of constructing the Belief MA of a PODFT: First, by constructing an adapted version of a Markov automaton and then transforming it into a Belief Markov Automaton (BMA) including the partial observability information. Since a BMA is also an MA, we can from there proceed using the previously established synthesis process.

In addition to the knowledge which nodes have currently failed, a PODFT also needs to memorize which of these failed nodes have been observed. We therefore extend the notion of a DFT state to a PODFT state by adding in a set of observed nodes.

Definition 6.2 (PODFT State). *A PODFT state s is a tuple*

$$s = (\text{history}, \text{claims}, \text{obs})$$

with $(\text{history}, \text{claims})$ being a DFT state and $\text{obs} \subseteq N(\mathcal{T})$ a set of observed failed nodes.

We denote the projection of the observable fragment of a PODFT state s with $O(s) := (\text{obs}(s), \text{claims}(s))$ where $\text{obs}(s) := \text{obs}$. For the BMA of a fully observable Markov automaton, we use probability distributions over PODFT states. Formally, this leads to the following definition:

Definition 6.3 (PODFT Belief-State). *A PODFT belief-state $\mathbf{b} \in \text{Dist}(S(\mathcal{A}))$ of a PODFT \mathcal{T} is a probability distribution over $S(\mathcal{A})$ with $\mathcal{A} := \text{MA}[\mathcal{T}]$ such that for any s, s' it holds that:*

$$\mathbf{b}(s) > 0, \mathbf{b}(s') > 0 \text{ iff } O(s) = O(s')$$

Fig. 6.5 shows an example of how fully observable MA states are represented in the partially observable BMA. In the fully observable case, there are two states (B_1B_3) and (B_2B_3) . The partially observable one may have any probability distribution over these two states. Here, we picked that with 25% we are in (B_1B_3) and 75% we are in B_2B_3 . In both cases, the only observed event is B_3 . Note that in the partially observable state we do not know what the concrete state is, as we only know the observation $O(B_3)$, however, we still have the full knowledge over the probability distribution. For the partially observable state space, this means that later the positional metrics need to be positional with regard to the partially observable state with the known distribution.

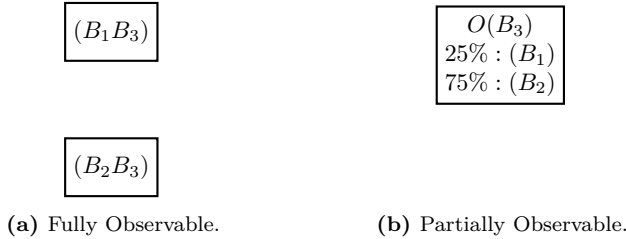


Fig. 6.5. Fully observable MA states vs partial observable MA states.

To label the observation transitions we will once again use sets of observations since multiple nodes may fail simultaneously. This may then lead to simultaneous observations by MONITOR gates. We formally define the set of all observation sets of a PODFT \mathcal{T} to be:

$$OS(\mathcal{T}) = \{O \subseteq \{n \mid n \in N(\mathcal{T})\} \cup \{n^r \mid n \in N(\mathcal{T})\}\}$$

First of all, note that the observation events are now based on the nodes in the tree instead of the basic events. Furthermore, we do not exclude the empty set, as we will later need it to model the lack of observations.

In the following, we will often require to know whether a node can be observed in a current state. To capture the set of monitors that can observe a fault tree

node n in state s , we define the set:

$$M(s, n) := \{m \mid T(\mathcal{T}, m) = t\text{-MONITOR}, P(n, m), m \notin \text{failed}(s)\}$$

Only monitors that have not failed in s are allowed membership to the set. Furthermore, we denote the set of immediate monitors that can observe n with:

$$M(s, n)_{t=0} := \{m \mid m \in M(s, n), t(m) = 0\}$$

Likewise, the following set defines the set of delayed monitors observing n :

$$M(s, n)_{t>0} := \{m \mid m \in M(s, n), t(m) > 0\}$$

Similarly to the closure of failed basic events $cl(s, b)$ under FDEPs, we define the closure under observation:

$$ocl(s, n) \subseteq \{n' \mid \exists m \in M(s, n)_{t=0}: P(n', m)\}$$

It denotes for a given failure of a node n in state s the set of transitively failed nodes that any immediate monitor can observe. Analogously, given a repair event n^r , we define the observation closure:

$$ocl(s, n^r) \subseteq \{n'^r \mid \exists m \in M(s, n)_{t=0}: P(n', m)\},$$

which contains all nodes that are transitively repaired and immediately observable.

For a PODFT belief state, we also need to revisit the notion of failure. In a DFT, failure corresponds to the occurrence of the TLE. However, in a PODFT, we may believe that the TLE has occurred but may not be certain of it. To quantify the probability that the TLE has failed in a given belief-state \mathbf{b} we define:

$$F(\mathbf{b}) = \sum_{tle \in \text{failed}(s)} \mathbf{b}(s)$$

In the following, we consider any state with $F(\mathbf{b}) > 0$ a potential fail state and $F(\mathbf{b}) = 1$ a fail state. Regarding the exit rate of a belief state \mathbf{b} , we define:

$$\text{exit}(\mathbf{b}) := \sum_{s \in S(\mathcal{A})} (\mathbf{b}(s) \cdot \text{exit}_{\mathcal{A}}(s))$$

Furthermore, since $O(s) = O(s')$ for any s, s' with $\mathbf{b}(s) > 0, \mathbf{b}(s') > 0$, we can define the short hand notation $O(\mathbf{b}) = O(s)$ for any s with $\mathbf{b}(s) > 0$ to reference the observable fragment of a belief state \mathbf{b} . Furthermore, given an event $e \in \{n, n^r\}$, either representing a failure or a repair event, we also define

the observation event $O(e)$ to capture the occurrence of observing e . Examples of belief state spaces can be found in Sec. 6.5 on page 119.

Using the PODFT state and the PODFT belief state, we can now define the two Markov Automata for the belief construction. We first start by defining the fully observable MA. The main two additions to the MA construction are:

- Observation events and corresponding observation transitions
- Updating the observation information

We formalize this in the PODFT-MA semantics below:

Definition 6.4 (PODFT-MA Semantics). *A PODFT-MA semantics is a mapping $MA[\Box]$ from an PODFT \mathcal{T} to an MA $\mathcal{A} := MA[\Box[\mathcal{T}]$.*

For the PODFT-MA semantics, we take over all construction rules for the NdDFT-MA semantics and extend them as follows: First, we introduce a new probabilistic state type \mathbb{O} that will represent that a probabilistic observation needs to be resolved. The updated labeling function thus has the updated codomain:

$$L : S \rightarrow 2^{\{M,N,P,\mathbb{O}\} \cup \{\text{FAIL}, \text{OP}\}}$$

For the initial state, we simply set $obs = \emptyset$. We next update the rules for generating state successors. Let $s = (history, claims, obs)$ be a state.

Updating Markovian successor generation Assume s is a Markovian state. For any enabled failure or repair event, generate the successors as usual.

In addition, let $enabled^O(s)$ denote the enabled delayed failure observations. That is, for any event $O(n) \in enabled^O(s)$, it holds that

- $n \notin obs(s)$ and $n \in failed(s)$ (n has failed but was not yet observed), and
- $M(s, n)_{t>0} \neq \emptyset$ (A delayed, operational monitor gate is observing n).

Generate the successor state $s' := (history, claims, obs \cup \{ocl(s, n)\})$.

For the case of observing a repair n^r , let $enabled^{Or}(s)$ denote the enabled delayed repair observations. That is, for any event $O(n^r) \in enabled^{Or}(s)$, it holds that

- $n \in obs(s)$ and $n \notin failed(s)$ (n is not failed but is currently observed as failed), and

- $M(s, n)_{t>0} \neq \emptyset$ (A delayed, operational monitor gate is observing n).

Generate the successor state $s' := (\text{history}, \text{claims}, \text{obs} \setminus \{\text{ocl}(s, n^r)\})$.

In both cases, for the transition, generate the Markovian transition

$$C(s, \text{ocl}(s, e) : \lambda_o, s'),$$

where

$$\lambda_o := \sum_{M(s,e)_{t>0}} t(m)$$

denotes the total observation rate, and $e \in \{n, n^r\}$ marks the event as a failure or repair event respectively. Finally, mark the successor state as non-deterministic by setting $L(s') := L(s') \cup \{\mathbb{N}\}$.

Updating probabilistic successor generation Next, consider $P \in L(s)$, that is, s is a probabilistic state. Note that with the new $\mathbb{0}$ state type that will come into play, the construction has two markings for probabilistic states. Here, we have the marking from the previous construction for resolving the occurrence of immediate basic events. After resolving the immediate basic events, we update the construction to resolve the immediate observers. For this purpose, the successor generation of P states remains mostly the same, except for an update to the labeling rule: Any successor state that would be labeled with $\{\mathbb{N}\}$ is instead labeled with $\{\mathbb{0}\}$.

Immediate observer successor generation Assume s is labeled with $\mathbb{0} \in L(s)$. Handling immediate observers is similar to handling Markovian states. Since the monitors do not have observation probabilities, all transitions will be simply labeled with probability 1. Note that in the BMA construction, the observation probabilities will no longer be 1 due to the beliefs.

We need to capture the set of all immediate observations in state s , including all repair observations and all fail observations. For this purpose we define the following two helper sets:

$$\begin{aligned} \text{unobs}^F(s) &:= \bigcup_{n \in N(\mathcal{T})} \{n \mid n \in \text{failed}(s), n \notin \text{obs}(s)\} \\ \text{unobs}^R(s) &:= \bigcup_{n \in N(\mathcal{T})} \{n^r \mid n \notin \text{failed}(s), n \in \text{obs}(s)\} \end{aligned}$$

$\text{unobs}^F(s)$ then captures the set of all outstanding failure observations and $\text{unobs}^R(s)$ the set of all outstanding repair observations. The total set of

outstanding observations is then:

$$unobs(s) := unobs^F(s) \cup unobs^R(s)$$

Generate the successor with the updated observation information:

$$s' := (history, claims, obs \cup unobs^F(s) \setminus unobs^R(s)),$$

and also generate the corresponding probabilistic transition $P(s, unobs(s) : 1, s')$. Finally, mark s' as a non-deterministic state by updating the labeling with $L(s') := L(s') \cup \{\mathbb{N}\}$.

For observation events, going from the MA to the BMA is now mostly a matter of grouping the corresponding states sharing equal observation sets. However, the construction becomes far trickier for regular failure and repair events. Since the events are only observed in the observation transitions, we lose the information about which failure or repair event actually happened. From an outside perspective, any enabled event could have occurred. In order to deal with this process, the construction will employ a special transition symbol τ representing the *silent occurrence* of a repair or failure event. The recovery automaton will not be able to directly respond to the τ -transitions since they are not observable. Instead, we will consider an enriched RA model capable of performing timeout transitions to estimate a decision point and switch strategies. This will represent that so much time has passed that it makes more sense to assume that a τ -transition occurred than not. In other words, the mean time to happen of the τ -transition will come into play. Since the belief state space is guaranteed to be finite, we add a ϵ criterion to the construction, where we only distinguish between two belief states if they significantly differ in belief. The significance level is here defined using ϵ and for the difference between two belief states we define the regular 2-norm $\|\mathbf{b}_1 - \mathbf{b}_2\| := \sqrt{\sum_s (\mathbf{b}_1(s) - \mathbf{b}_2(s))^2}$. Overall, this gives us the epsilon criterion that we only distinguish between $\mathbf{b}_1, \mathbf{b}_2$ iff $\|\mathbf{b}_1 - \mathbf{b}_2\| > \epsilon$.

Definition 6.5 (PODFT-BMA Semantics). *A PODFT-BMA semantics is a mapping $BMA \llbracket \llbracket \epsilon$ from an PODFT \mathcal{T} to an MA $\mathcal{B} := BMA \llbracket \llbracket \mathcal{T} \rrbracket \epsilon$.*

In the following, we give the BMA construction. To generate the successors, we distinguish between three processes of successor state generation:

- From a delayed observation event.
- From an unobservable fault event represented as a single silent event τ .

- From an immediate observation event.

Consider some fully observable Markov automaton as produced by the updated PODFT-MA semantics:

$$\text{MA}[\mathcal{T}] = \mathcal{A} = (S, L, A, N, C, P, s_0)$$

We then define the PODFT-BMA:

$$\text{BMA}[\mathcal{T}] = \mathcal{B} = (B, L', A', N', C', P', b_0)$$

As initial state, generate $\mathbf{b}_0(s_0) := 1$ and $\mathbf{b}_0(s) = 0$ for any other state $s \in S$.

Now, let \mathbf{b} be some generated belief state. Then, set $L(\mathbf{b}) = L(s)$ for any s with $\mathbf{b}(s) \neq 0$. If there already exists a prior belief state \mathbf{a} such that $\|\mathbf{a} - \mathbf{b}\| < \epsilon$ and $L(\mathbf{a}) = L(\mathbf{b})$, then reroute all incoming transitions to b to a instead, and then discard \mathbf{b} . Otherwise, continue the construction. For non-deterministic states $N \in L(\mathbf{b})$, we simply copy over the non-deterministic transitions and likewise copy the beliefs to the corresponding successor states. For the other state types, we distinguish between three cases.

Delayed Observation Successor Assume in the following that $M \in L(\mathbf{b})$. Let $C(s, O, s')$ be some observation transition for some state s into another state s' with $\mathbf{b}(s) \neq 0$. Analogously to the basic event failures in the fully observable MA, generate a successor \mathbf{b}' for the delayed observation events O . And generate a Markovian transition $C'(\mathbf{b}, O : \lambda_o, \mathbf{b}')$ with transition rate λ_o . The transition rate is affected by two factors: The observation rates of the observing MONITOR gates and the belief that we are in a state where n is failed. Finally, let

$$S_n = \{u' \mid C(u, O, u'), \mathbf{b}(u) \neq 0\}$$

be the successor states sharing the same observations as s' . Let $M_n = M(s, n)_{t>0}$ be the set of non-failed, delayed monitors observing n . Note that we introduced the semantic restrictions of monitor failure always being observable. Therefore, M_n is consistent over all beliefs in \mathbf{b} . Then λ_o is defined as:

$$\lambda_o := (\sum_{s \in S_n} \mathbf{b}(s)) \cdot (\sum_{m \in M_n} t(m))$$

For any belief $u \notin S_n$ not compatible with the new observation set, we set $\mathbf{b}'(u) = 0$. The freed probability mass is then distributed over the remaining beliefs. We therefore obtain for any $u \in S_n$:

$$\mathbf{b}'(u) = \mathbf{b}(u) / \sum_{u' \in S_n} \mathbf{b}(u')$$

Unobservable τ -Successor Assume in the following that either $\mathbf{M} \in L(\mathbf{b})$ or $\mathbf{P} \in L(\mathbf{b})$. Assume further that \mathbf{b} has at least one enabled fault event. Define $\lambda_{\mathbf{b}} := \text{exit}(\mathbf{b})$ and $T_{\mathbf{b}} := 1/\lambda_{\mathbf{b}}$. In the BMA, generate a τ -transition and a successor state \mathbf{b}' . A τ -transition models the belief change after one mean time to happen $T_{\mathbf{b}}$ for any enabled Markovian event. There is a probability p_{remain} that no Markovian event fired in this time frame and we maintain our prior beliefs from \mathbf{b} . This yields:

$$p_{\text{remain}}(\mathbf{b}) := e^{\lambda_{\mathbf{b}} \cdot T_{\mathbf{b}}} = e^{-1}$$

The probability mass p_{exit} for a Markovian event firing is therefore:

$$p_{\text{exit}}(\mathbf{b}) := (1 - p_{\text{remain}}(\mathbf{b})) = 1 - e^{-1}$$

Note that both probabilities are constants due to the probability mass change after one mean time to happen being constant e^{-1} , independently of the exit rate. With this, the update rule for the belief change of a τ -transition can be defined as following:

$$\begin{aligned} \mathbf{b}'(s) &= \mathbf{b}(s) \cdot p_{\text{remain}}(\mathbf{b}) + p_{\text{exit}}(\mathbf{b}) \cdot \sum_{C(s_p, \lambda_{s_p}, s)} \mathbf{b}(s_p) \lambda_{s_p} / \lambda_{\mathbf{b}} \\ &= \mathbf{b}(s) \cdot e^{-1} + (1 - e^{-1}) \cdot \sum_{C(s_p, \lambda_{s_p}, s)} \mathbf{b}(s_p) \lambda_{s_p} / \lambda_{\mathbf{b}} \end{aligned}$$

For the transition rate of the τ -transition, we cannot directly use the regular exit rate. This is due to the Markovian transition modeling a guaranteed transition into a successor state. However, the belief transitioning semantics models two simultaneous probabilistic processes: In addition to the exponentially delayed distribution of eventually performing a transition, we also have the probability p_{prob} to not trigger a Markovian transition at all. Therefore, the exit rate has to be conditioned to the remaining probability p_{exit} . This gives the transition rate of the τ -transition and also the adjusted exit rate of the belief state b .

$$\text{exit}(\mathbf{b})_{\tau} = \lambda_{\mathbf{b}} / p_{\text{exit}}(\mathbf{b}) = \lambda_{\mathbf{b}} / (1 - e^{-1})$$

Immediately observable events Assume finally $\mathbf{0} \in L(\mathbf{b})$. We proceed similarly to the case of the delayed observation transitions. Let $P(s, O : 1, s')$ be some immediate observation transitions for some state s into another state s' with $\mathbf{b}(s) \neq \mathbf{0}$ and observation set O . Generate a successor belief state \mathbf{b}' for the immediate observation event O with a probabilistic transition $P'(\mathbf{b}, O : p_O, \mathbf{b}')$ with transition probability $p_O := \sum_{P(u, O : 1, u'), \mathbf{b}(u) \neq \mathbf{0}} \mathbf{b}(u)$. Let further

$$S_O = \{u' \mid P(u, O, u'), \mathbf{b}(u) \neq 0\}$$

be the successor states sharing the same observations as s' . For any belief $u \notin S_O$ not compatible with the new observation set, we set $\mathbf{b}'(u) = 0$. For all other states we redistribute the probability mass and obtain for any $u \in S_O$:

$$\mathbf{b}'(u) := \mathbf{b}(u) / \sum_{u' \in S_O} \mathbf{b}(u')$$

The construction of state spaces with τ -transitions is abstractly depicted in Fig. 6.6.

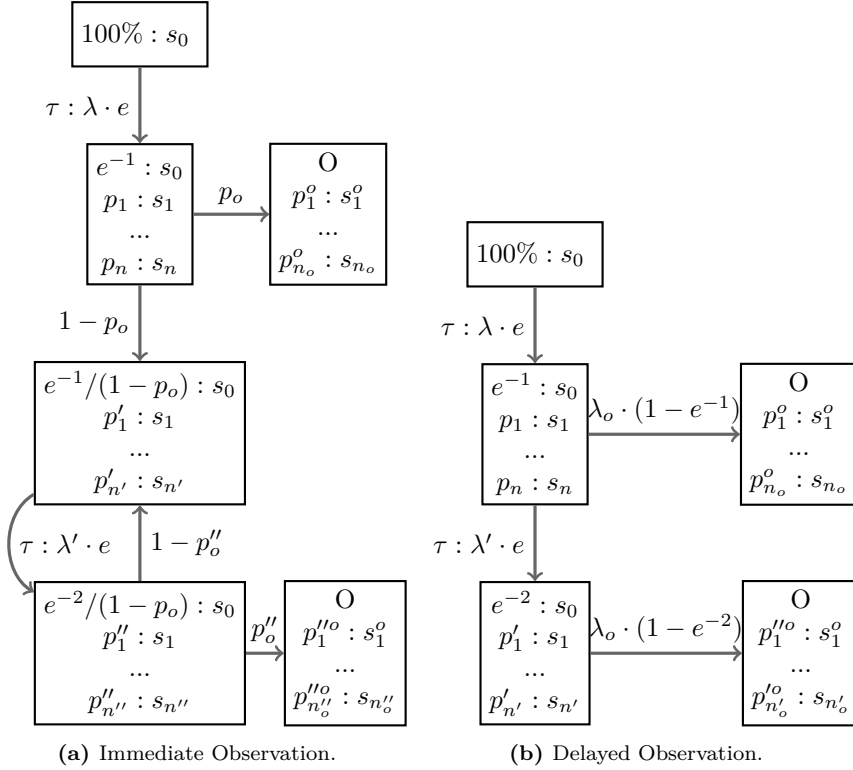


Fig. 6.6. State space scheme for τ -Transitions with immediate and delayed observers.

In each example, there is one observer: an immediate observer and a delayed observer with observation rate λ_o . The exit rate of a belief state is denoted

with λ, λ' respectively. For the immediate observer, the observation probability $p_o = \sum_{i=1}^n p_i$ corresponds to the total probability of being in an observable state. With n_o, n'_o , we denote the number of states that belong to the belief state after the observation. For the state space, ϵ is chosen such that

$$|e^{-1}/(1 - p_o) - e^{-2}/((1 - p_o)(1 - p''_o))| < \epsilon$$

and $|e^{-2} - e^{-3}| < \epsilon$. That is, in the case of the immediate observer, we stop after two τ -transitions and get a loop back. In the case of delayed observation, there is no explicit loop, as the successor of the second τ -transition is a Markovian state. Due to the ϵ criterion, a third τ -transition would produce a self-loop, which can be ignored for CTMCs. In both cases, the state space construction can stop after two τ -transitions. Note that with every τ -transition, the probability of being in s_0 decreases, and the total probability mass of being in some other state subsequently increases. For simplicity, repair events are not considered in these examples. The states with observation O are non-deterministic and would be followed by recovery actions.

It follows from the construction that \mathbf{b}' is still a distribution and does not inflate the total probability mass over 1. We show this in the following proof.

Proposition 6.1. *Let \mathbf{b} be a belief state and \mathbf{b}' a generated τ -transition successor. Then $\sum_s \mathbf{b}'(s) = 1$.*

Proof. Consider \mathbf{b}, \mathbf{b}' as required. Then

$$\begin{aligned} \sum_s \mathbf{b}'(s) &= \sum_s (\mathbf{b}(s) \cdot e^{-1} + (1 - e^{-1}) \cdot \sum_{C(s_p, \lambda_{s_p}, s)} \mathbf{b}(s_p) \lambda_{s_p} / \lambda_{\mathbf{b}}) \\ &= e^{-1} \cdot \sum_s \mathbf{b}(s) + \sum_s ((1 - e^{-1}) \cdot \sum_{C(s_p, \lambda_{s_p}, s)} \mathbf{b}(s_p) \lambda_{s_p} / \lambda_{\mathbf{b}}) \\ &= e^{-1} \cdot \sum_s \mathbf{b}(s) + (1 - e^{-1}) \cdot \sum_s (\sum_{C(s_p, \lambda_{s_p}, s)} \mathbf{b}(s_p) \lambda_{s_p}) / \lambda_{\mathbf{b}} \\ &= e^{-1} + (1 - e^{-1}) \cdot \sum_s (\sum_{C(s_p, \lambda_{s_p}, s)} \mathbf{b}(s_p) \lambda_{s_p}) / \lambda_{\mathbf{b}} \quad (\sum_s \mathbf{b}(s) = 1) \\ &= e^{-1} + (1 - e^{-1}) \cdot \sum_s (\mathbf{b}(s) \text{exit}(\mathbf{b})) / \lambda_{\mathbf{b}} \\ &= e^{-1} + (1 - e^{-1}) \cdot \lambda_{\mathbf{b}} / \lambda_{\mathbf{b}} \\ &= 1 \end{aligned}$$

□

6.3 Partially Observable Recovery Automaton

Due to the addition of the silent τ -transitions in the Belief MA, the old formalism for RA's is no longer strong enough to express the optimal recovery behavior.

Since the original RA definition hinges on observing events, the RA cannot change its recovery behavior based on an unobserved τ -transition.

Consider the following variation of the memory system illustrated in Fig. 6.7. Here, a τ -transition may change the optimal recovery strategy.

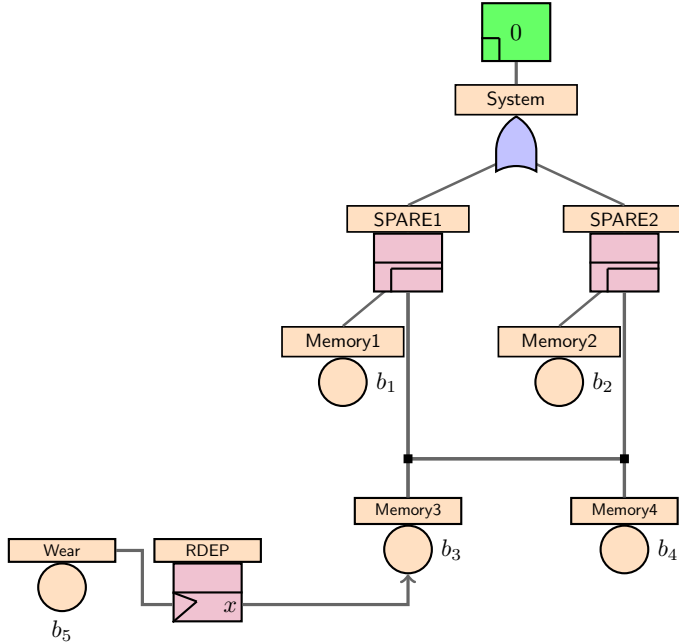


Fig. 6.7. Example PODFT with RDEP. The more time passes, the more likely it becomes for Wear to occur and make Memory3 the sub-optimal spare choice.

In the fully observable case, the RA would choose the spare memory with the lowest failure rate. Consider for this example $F(b_3) < F(b_4)$. Then claiming b_3 would be prioritized over claiming b_4 . However, the PODFT also has an RDEP modifying the failure rate of b_3 . If $x \cdot F(b_3) > F(b_4)$, then upon b_5 occurring, it would be better for the RA to switch strategies and prioritize claiming b_4 . However, in this case, the wear event is not observable and therefore creates a silent τ -transition. Instead of relying on the knowledge of the wear event occurring, the RA now needs to rely on the belief of the wear event having occurred.

To compensate for this, the basic RA model needs to be extended. The idea is to add a time-based transition in the RA based on the mean time to happen for the silent τ -transition. This could be achieved by elevating the RA to a higher automaton class such as Timed Automata (TA). However, this semantic elevation would have severe consequences on the composition between NdDFT and RA since we would now need a model that both encompasses the Markovian nature of MA's and the timed nature of TA's. While there are approaches to compose continuous time MA's with TA's [70], we can perform the following simplification: Instead of introducing the whole concept of clocks present in TAs, we only add a single timeout transition for each RA state. This timeout transition then corresponds to the silent τ -transition. For the composition operation, this timeout transition can then reversely be approximated using a Markovian transition.

As for the concept of recovery strategies, we can now adapt the fully observable case by simply exchanging the input set to any observation set.

Definition 6.6 (Partially Observable Recovery Strategy). *A recovery strategy for an PODFT \mathcal{T} is a mapping $Recovery : OS(\mathcal{T})^* \rightarrow RS(\mathcal{T})^*$ such that*

- $Recovery(\varepsilon) = \varepsilon$ and
- $Recovery(O_1, \dots, O_n) = Recovery(O_1, \dots, O_{n-1}), rs_n$ with $rs_n \in RS(\mathcal{T})$.

Putting these ideas into a concrete definition gives us the following simplified concept of a Partially Observable Recovery Automaton (PORA).

Definition 6.7 (Partially Observable Recovery Automaton). *A Partially Observable Recovery Automaton (PORA) $\mathcal{R}_{\mathcal{T}} = (Q, \delta, t, q_0)$ of a PODFT \mathcal{T} is an automaton where*

- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- $t: Q \rightarrow Q \times \mathbb{R}_{>0}$ is a deterministic timeout transition function that maps the current state to the successor state and the amount of time that needs to pass, and
- $\delta: Q \times OS(\mathcal{T}) \rightarrow Q \times RS(\mathcal{T})$ 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 transition function δ is extended to $\delta^* : Q \times OS(\mathcal{T})^* \rightarrow RS(\mathcal{T})^*$ by letting

$$\begin{aligned} \delta^*(q, \epsilon) &:= \epsilon \\ \delta^*(q, O \cdot w) &:= rs \cdot \delta^*(q', w) \text{ with } \delta(q, O) = (q', rs) \end{aligned}$$

for any $q \in Q$, $O \in OS(\mathcal{T})$ and $w \in OS(\mathcal{T})^*$. The recovery strategy induced by a recovery automaton \mathcal{R} , $Recovery_{\mathcal{R}} : OS(\mathcal{T})^* \rightarrow RS(\mathcal{T})^*$, is given by $Recovery_{\mathcal{R}}(w) := \delta^*(q_0, w)$.

To visually distinguish the transitioning processes between PORA and RA we introduce a differing notation for labeling the event guards. For the PORA we write $O(F(n))$ for a failure observation of n and $O(R(n))$ when observing a repair event n^r .

6.4 Synthesis Workflow

With the adoption of PORAs, the synthesis goal now naturally changes to creating PORAs from PODFTs. In the next sections, we investigate which adaptations need to be made to the individual steps of the synthesis workflow.

6.4.1 PORA Extraction

Since we have reduced the problem of PORA synthesis again to a schedule optimization problem on a Markov automaton, the extraction process of the PORA is mostly analog to the extraction of the RA in the fully observable case. However, before, we only had to deal with transitions that were strictly of the form of an event occurring and then being followed by a sequence of recovery actions. Now, we also need to deal with the silent τ -transitions, which in the PORA will become timeout transitions. We extend the PORA extraction procedure as follows:

Consider a τ -transition starting in some belief state \mathbf{b} , resolving immediate observations in a probabilistic belief state \mathbf{p} , and then transitioning into a successor belief state \mathbf{b}' . According to the state space construction, this sequence then has the following form:

$$\mathbf{b} \xrightarrow{\tau:\lambda} \mathbf{p} \xrightarrow{1-p_o} \mathbf{b}'$$

The state \mathbf{p} potentially has an observation transition for any immediately observable node o_i . On the other hand, \mathbf{b} and \mathbf{b}' are Markovian and may have

Markovian observation transitions for each delayed observable node o_j . Generate the timeout transition:

$$t(\mathbf{b}) = (\mathbf{p}, 1/(\lambda \cdot e))$$

Furthermore, generate the transition $\delta(\mathbf{b}, \emptyset) = \mathbf{b}'$ representing not-observing any observable event. Finally, for the observation case, proceed as usual. By construction, for any enabled set of observations O , there exists a state sequence of recovery actions of the form:

$$\mathbf{p} \xrightarrow{O:p} \mathbf{b}_1 \xrightarrow{r_1} \mathbf{b}_2 \xrightarrow{r_2} \dots \mathbf{b}_{n-1} \xrightarrow{r_{n-1}} \mathbf{b}_n$$

Generate the transition $\delta(\mathbf{p}, O) := (\mathbf{b}_n, r_1 \dots r_n)$. From \mathbf{b}_n , the construction then proceeds as usual.

In the case that an observation O occurs before the timeout, the recovery action knows that a silent τ -transition occurred and needs to perform the appropriate recovery actions. Consider a belief state \mathbf{b} with no defined outgoing transition for an event O . Then let \mathbf{b}' be the state reachable with a minimal number of transitions with a defined outgoing transition for the observation O . Copy the behavior by defining:

$$\delta(\mathbf{b}, O) := \delta(\mathbf{b}', O)$$

6.4.2 PORA and MA Synchronization

Since PORA and MA both operate on observation events, the synchronization is mainly reduced to the same case as the RA and MA synchronization. Not handled are the new τ -transitions. In order to consider them in the synchronization process, we extend the \parallel operator as follows:

Let (\mathbf{b}, q) be a state generated in the synchronized product $\mathcal{C} := \mathcal{R} \parallel \mathcal{B}$ of the RA \mathcal{R} and the BMA \mathcal{B} , such that \mathbf{b} is a belief state with a τ -transition and q a state in the recovery automaton. By construction, q then has a timeout transition $t(q) = (\mathbf{b}, t)$ for some $t \in \mathbb{R}$, and \mathcal{B} has a sequence of the form:

$$\mathbf{b} \xrightarrow{\tau:\lambda \cdot e} \mathbf{b}$$

In \mathcal{C} , generate the Markovian transition $((\mathbf{b}, q), \lambda \cdot e, (\mathbf{p}, \mathbf{p}))$. For any successor of \mathbf{p} triggered by some observation set O , we proceed as usual. In the case of not observing any event, that is, for the transition $\delta(\mathbf{p}, \emptyset) = q'$ for some q' , we also, by construction, have in the BMA the probabilistic transition:

$$\mathbf{p} \xrightarrow{\tau:1-p_o} \mathbf{b}'$$

To synchronize these two transitions, generate in \mathcal{C} the Markovian transition:

$$((\mathbf{p}, \mathbf{b}), 1 - p_o, (\mathbf{b}', q'))$$

Note that the timeout transition information itself is technically redundant in the construction. The timeout itself $t = 1/(\lambda \cdot e)$ and the target state of the timeout transition are known to the construction. An explicit representation of the timeout transition within the automaton makes it more understandable to a human. Furthermore, it allows performing some minor minimization techniques for merging successive timeout transitions, which we will briefly discuss in the implementation details.

6.4.3 Orthogonality under Partial Observability

Partial observability also affects the reduction of orthogonal states. Originally, each non-repairable BE can occur at most once. However, in a PODFT, even non-repairable events can be observed multiple times. Consider, for an example, the simple PODFT with a SPARE and an immediate MONITOR gate, depicted in Fig. 6.8.

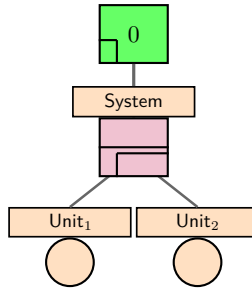


Fig. 6.8. Repeated failure observations. The event System can be observed twice.

If a CLAIM(System, Unit₂) is performed upon observing the failure of System, and if Unit₂ fails, then it is possible to observe the sequence:

$$\{O(F(System))\}\{O(R(System))\}\{O(F(System))\}$$

Since the invariant of only each failure occurring at most once is no longer fulfilled, orthogonal minimization is not applicable. We therefore disable the orthogonal minimization on PORAs.

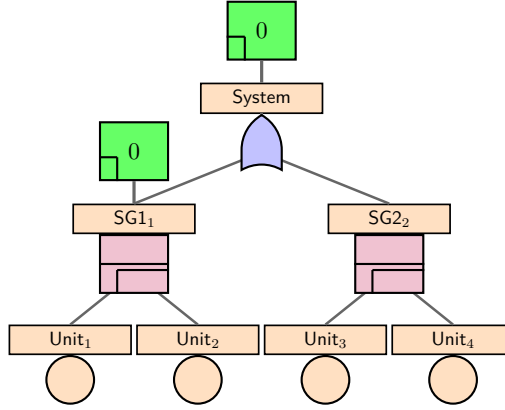


Fig. 6.9. Example of a non-modularizable PODFT. Only by observing *System* and *SG₁* together is it possible to deduce that an event occurred in the *SG₂* sub-tree.

6.4.4 Adapting Modularization

In order to lift our results from the fully observable case, we now also need to respect the flow of observation information. Consider the following PODFT example, depicted in Fig. 6.9, where two sub-trees would theoretically each form a module but cannot be modularized. Note that in the fully observable case *SG₁* and *SG₂* would each form a module. However, in the partially observable case, the observation of *SG₁* has an influence on the recovery behavior in *SG₂*. When observing only $\{O(F(System))\}$, we can deduce that the fault must have occurred in the *SG₂* sub-tree. On the other hand, observing $\{O(F(System)), O(F(SG_1))\}$ implies that a fault occurred in the *SG₁* sub-tree. Overall, we can conclude that the observation behavior of the two sub-trees is linked and cannot be considered separately.

While modularization becomes impossible in many cases, it is not generally impossible. The following examples, shown in Fig. 6.10 on the next page, demonstrate cases where we have partially observable modules, and modularization is indeed possible.

Adapting the modularization criteria requires the incorporation of the flow of observation information:

- The root of a module must be immediately observable, or
- All inputs to a SPARE gate must be immediately observable.

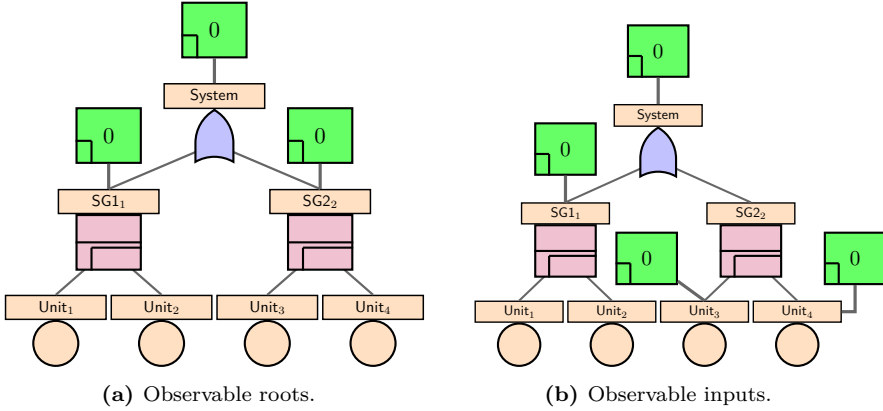


Fig. 6.10. Example of modularizable PODFTs. Since sufficient information to synthesize the optimal PORA is available, SG_1 and SG_2 form module roots.

6.4.4.1 Classifying modules

In the course of this work, we have now introduced multiple workflows to solve the RA synthesis problem for different types of fault trees with different levels of semantic power. This raises the question of whether the entire workflows need to be used for all modules of a PODFT. Originally, a key idea for modularization was to solve static modules using a dedicated algorithm for static FT analysis and dynamic modules using state-based approaches. We extend this idea for PODFTs and introduce the following module types

- Deterministic
- Non-Deterministic and fully observable
- Non-Deterministic and partially observable

If all basic events in a module are fully observable, we treat the module as the module of an NdDFT. As before, if a module contains no non-determinism, independently of whether there are partially observable events or not, then it is trimmed for the purpose of RA synthesis.

6.5 Synthesis Examples

To conclude the chapter on observability, we review the new objects and workflows by presenting and discussing some examples. We focus on giving the PODFT, an illustrative fragment of the Belief MA, and the synthesized PORA.

6.5.1 Probabilistic Claim Success

In a fully observable Markov Automaton, recovery actions (CLAIM, FREE) were always guaranteed to succeed. However, under partial observability, this is no longer guaranteed! For a simple example, we consider the PODFT and the corresponding BMA fragment shown in Fig. 6.11.

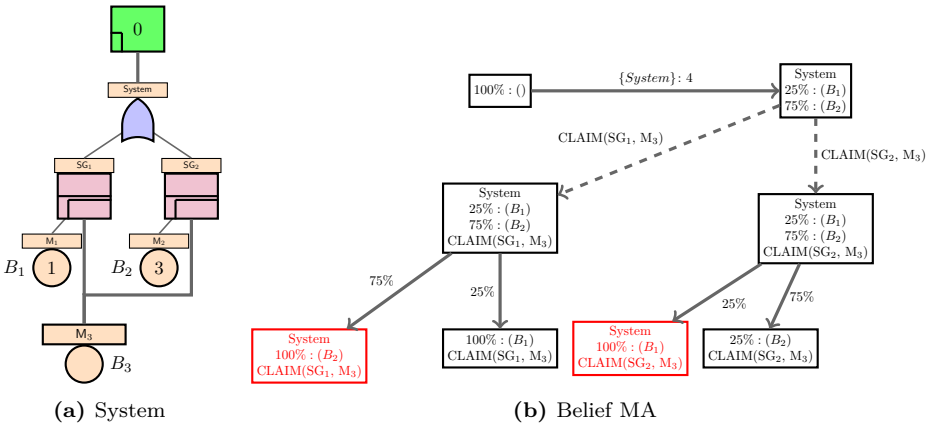


Fig. 6.11. Probabilistic Success of CLAIM action. Claiming M_3 with SG_2 has the highest probability of recovering System.

Observe how only the failure of System can be observed. However, since there are two failure sources, B_1 and B_2 , it is not evident which caused the top-level event to fail. Claiming the shared redundancy M_3 with SG_1 or SG_2 , respectively, is no longer guaranteed to repair the System! In this case, since the failure rate of B_2 is higher than the failure rate of B_1 , it is more likely for B_2 to be the source of failure and hence, claiming with SG_2 is the optimal move. This is reflected in the synthesized RA illustrated in Fig. 6.12 on the next page.

In case of the TLE failing, the spare gate with the higher likelihood of failure, that is, SG_2 gets to claim M_3 . If the TLE can still be observed or is observed in

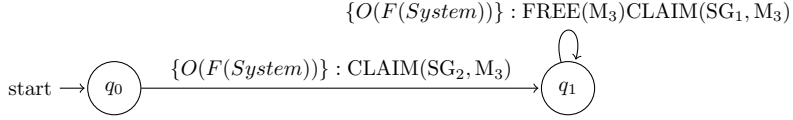


Fig. 6.12. Synthesized Recovery Automaton retrying CLAIM action.

the future due to another component failure, the spare is freed, and the SPARE gate SG_1 gets to perform its claim.

6.5.2 Delayed Monitor

We consider here an example of how a delayed MONITOR gate can affect the state space and synthesis. The PODFT illustrated in Fig. 6.13 has two MONITOR gates: An immediate one monitoring System and a MONITOR gate delayed by 1 time unit observing $Unit_1$. In other words, additional information on the failure of the TLE may be provided at a later point in time.

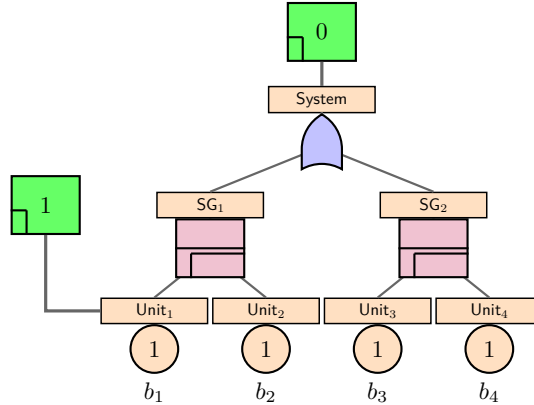


Fig. 6.13. System with delayed MONITOR. A delayed MONITOR gate observes $Unit_1$ with a time delay of 1.

The Belief MA fragment illustrated in Fig. 6.14 on the following page shows how this affects the corresponding belief state space. Upon observing the TLE, it is unknown which primary caused the failure. To avoid entering a fail state with a 50% probability, both SPARE gates need to perform a claim. When

claiming only one redundant unit, it is possible to enter a fail state. Upon learning of the failure from Unit₁, freeing Unit₂ leads guaranteed to a fail state. However, performing a FREE on Unit₄ still leaves the system with a 50% chance of survival. On the other hand, if Unit₁ is not observed, there is no guarantee as to which claim should be freed.

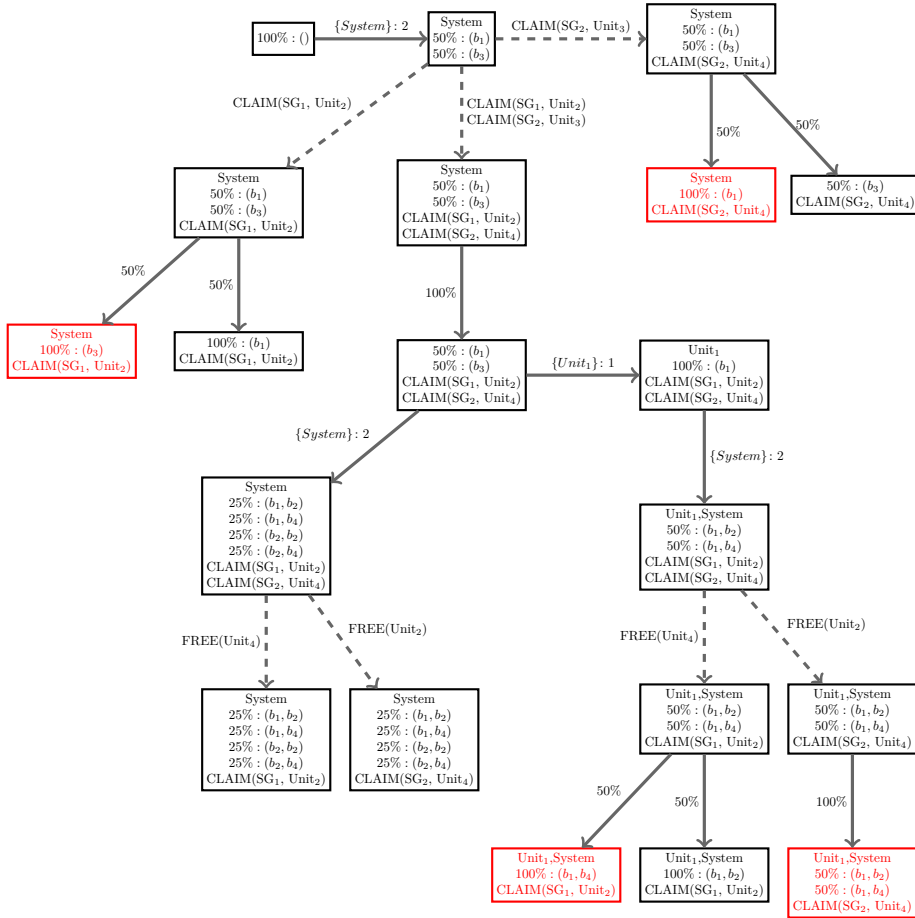


Fig. 6.14. Belief MA fragment for a system with a delayed MONITOR gate. Observing the delayed occurrence of Unit₁, gives additional information leading to a save recovery strategy by freeing Unit₄ upon the second System failure.

The synthesized recovery automaton is given in Fig. 6.15. If the delayed MONITOR provides no observation, the PORA is forced to attempt several claim/free actions in order to uncover the failure mode of the TLE. Observing that Unit₁ causes the TLE to occur allows the PORA to resolve the uncertainty in the state space and therefore has no need to attempt several claim configurations. Knowing that Unit₁ caused the earlier TLE event allows us to infer that Unit₃ is functional. Upon observing another occurrence of the TLE, recovery can then be achieved by freeing Unit₄ and switching back to Unit₃. Note that if the TLE is caused again by a unit belonging to SG₁, then the system is permanently failed as both units are no longer functional.

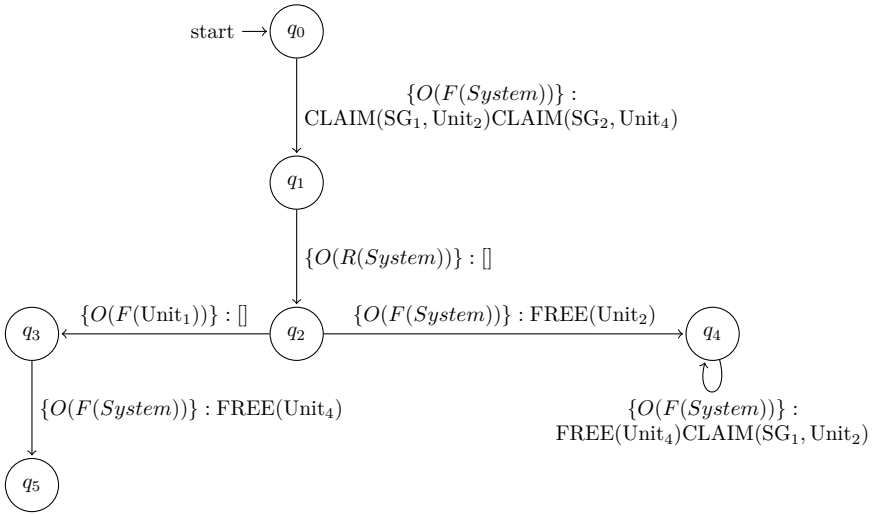


Fig. 6.15. Synthesized Recovery Automaton for the delayed monitor example. The observation of Unit₁'s failure simplifies the recovery.

6.5.3 Failable Monitor

We consider a variation of the previous example case where the MONITOR gate can fail. Fig. 6.16 on the following page shows the modified PODFT. Here, both monitors are immediate, but the second monitor gate now has a fault input.

Since SG₁ can be initially explicitly observed, the PODFT has, effectively, perfect knowledge of its current state. Since a failure in the left SPARE system

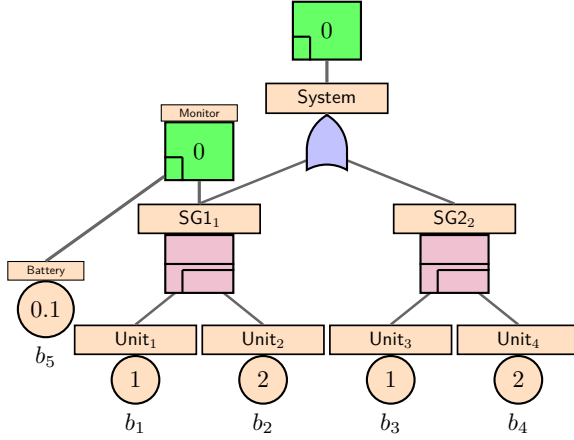


Fig. 6.16. Partially observable SPARE system with a failable MONITOR gate. SG_1 is monitored by an observer that may fail when b_5 occurs.

can be immediately observed, the exact failed basic event can be derived. Furthermore, only observing the TLE failure but not observing anything within the observable spare system also reveals that a BE in the right spare system must have failed. However, upon the failure of Monitor, this knowledge is subsequently lost.

The BMA fragment shown in Fig. 6.17 on the next page illustrates how the change of knowledge from the loss of the second monitor affects the decision process. The fragment depicts the very first basic event failure and the basic event failure after a monitor fails. We can see in the fragment that the states correspond to the regular MA while both monitors are operational. The optimal claim action can be trivially derived since the MA state is known with 100% certainty. In the case of monitor failure, the BMA has to deal with the uncertain superposition. In order to guarantee the System's function, both SPARE gates need to be switched to their redundancy. The case is similar to fragments of previous examples, so we do not further unwind the state space.

The synthesized RA is given in Fig. 6.18 on page 125. We can find the structure of the BMA fragment within the PORA. If the failure of Monitor is observed, the RA switches to a branch similar to the delayed monitor RA from Fig. 6.15 on the previous page, without the delayed information.

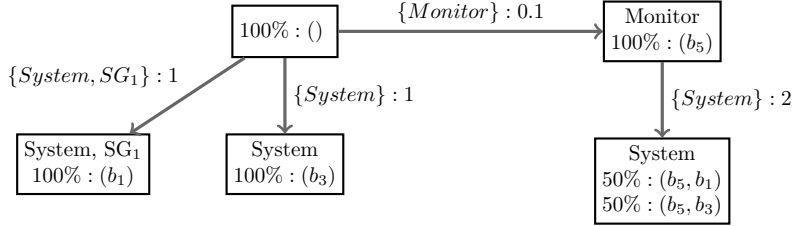


Fig. 6.17. Fragment of the BMA for the system with a failable MONITOR gate. When the MONITOR gate fails, the beliefs after observing a System failure become fuzzy.

As discussed earlier, the RA needs to claim both spares and if another TLE occurs, potentially attempt several claim configurations. If the monitor fails after the first unit fails, the PORA knows which unit needs to be claimed to recover from the first fault and can subsequently still act optimally. Note that due to the OR connection between the two SPARE subsystems a double failure in the same subsystem leads to total system failure. Therefore, if we know that the first failure occurred in a particular subsystem, then a second failure means total system failure. Performing a claim in the other subsystem is therefore the optimal choice, which is reflected in the decision process of the PORA.

6.5.4 Timeout Transitions

For the last example, we reconsider a simplified version of the memory system running example in Fig. 6.19 on page 126. The aim is to showcase an example with a silent τ -transition and how it propagates into a timeout transition in the PORA. In the PODFT, initially, Memory₃ is the favorite spare choice. However, due to the RDEP potentially increasing the failure rate of Memory₃, Memory₄ might become the better choice over time.

A fragment of the BMA for $\epsilon = 0.2$ is given in Fig. 6.20 on page 127. Values are rounded to two decimals after the comma. Note that in each transition state, the Markov automaton only assumes one event to have occurred, thus resolving the first uncertainty with a 100% chance of being in the state where only Memory₁ has failed. The low chance that this is the case is represented in the low transition rate. The state is not further expanded upon since it corresponds to a regular failure without uncertainty. Once two unobservable τ -transitions have occurred, the Belief MA is in a state where either just Memory₁

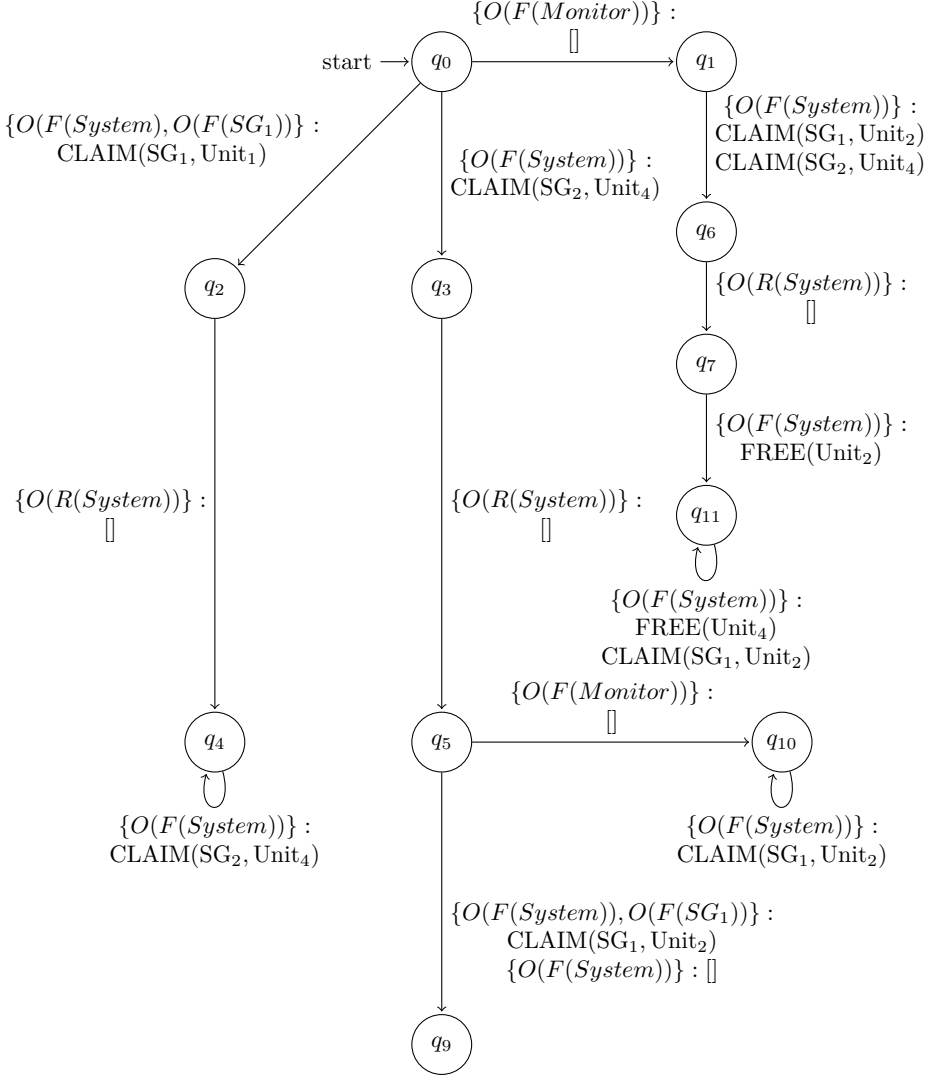


Fig. 6.18. Synthesized recovery automaton for a system with failable MONITOR gate.

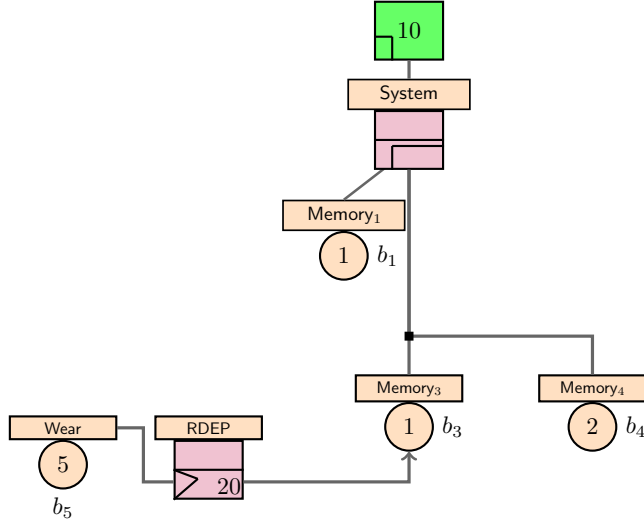


Fig. 6.19. Simplified PODFT of memory system with spare pool. The silent failure of Wear could make Memory₄ the better spare choice.

has failed or where additionally the RDEP has been triggered. For the claims, the empty recovery action \square is not illustrated since it clearly provides no benefit in this scenario. Note that after performing the claims, there are two possible events: The monitor observes the system to be functional again, or another silent τ -transition occurs, possibly triggering the RDEP or the claimed spare. Since the state space quickly increases in size, we stop the fragment illustration there.

The synthesized PORA is shown in Fig. 6.21 on the following page. As discussed above, initially, the PORA claims Memory₃ upon witnessing a failure. The timeout transition between q_0 and q_2 initiates a change of strategy. At this point, the PORA will prefer claiming Memory₄ over Memory₃. Note that the timeout delay accumulates the reciprocates of the failure rates shown in the BMA. This gives the timeout in the PORA $1/9.49 + 1/5.16 \approx 0.105 + 0.194 \approx 0.3$.

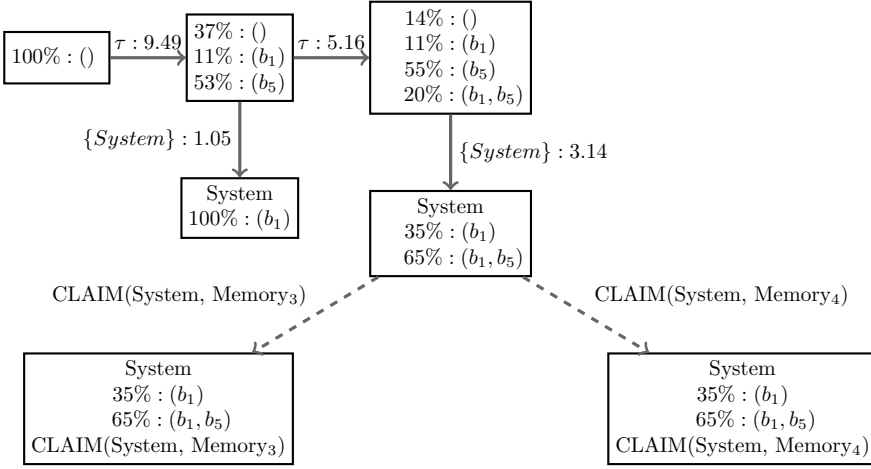


Fig. 6.20. Fragment of the Belief MA with a silent RDEP.

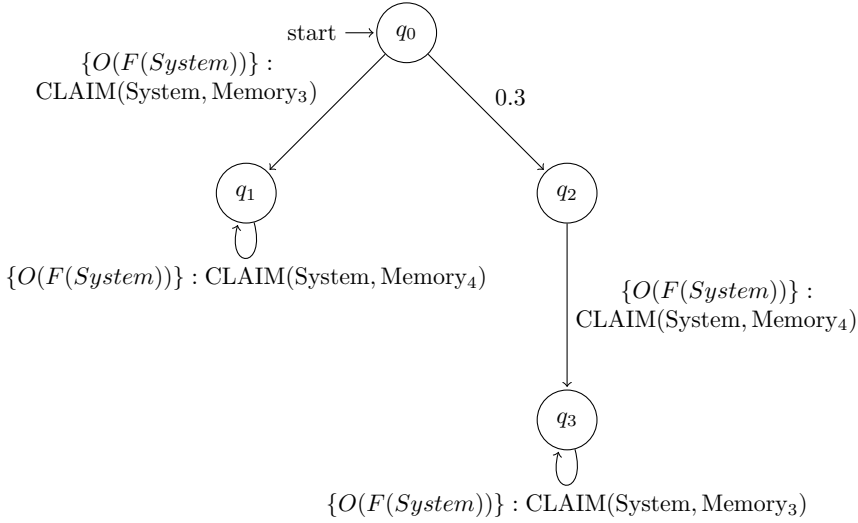


Fig. 6.21. Synthesized recovery automaton with timeout transition. If in state q_0 no observation occurs within 0.3 time units, the automaton switches to q_2 .

Chapter 7

Implementation

In this chapter, we discuss a prototype implementation of the described FDIR models and synthesis algorithms. We have implemented them within the Virtual Satellite 4.0 framework [71] which we will also further discuss in the following. Virtual Satellite 4 (VirSat) is an Eclipse-based framework intended for performing Model-Based Systems Engineering (MBSE) over the whole life cycle of space systems. VirSat provides a Generic Systems Engineering Language (GSEL) in which model extensions called Conceptual Data Models (CDMs) or just *concepts* can be described. Each concept addresses a specific engineering aspect such as system decomposition, interface management, budgeting of mass allocation and power consumption, and so on. The developed prototype is a VirSat application called VirSat FDIR [4, 72]. It provides such a concept for modeling the FDIR engineering aspect. This chapter discusses the conceptual data models, how they integrate into the data models of VirSat, implementation details of the state space representation and the synthesis algorithm, and further details on how the synthesis workflow can be used within the prototypical implementation.

7.1 Virtual Satellite 4 Framework

In the past years, much effort has been invested into enabling Model-Based Systems Engineering (MBSE) for the whole life cycle of a spacecraft. Part of these efforts is Virtual Satellite.

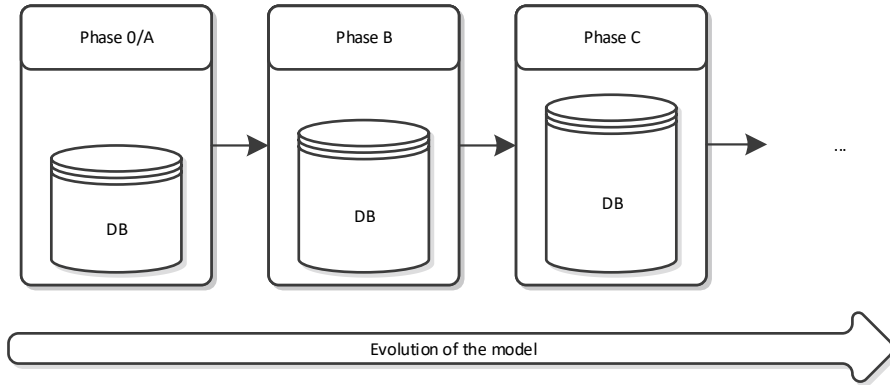


Fig. 7.1. Virtual Satellite database growing along the phases

VirSat is a concurrent engineering tool used at the Concurrent Engineering Facility (CEF) at the German Aerospace Center (DLR). It is also a software framework that allows for integrating various engineering processes across the individual phases of spacecraft design and operation and the different disciplines. The framework implements an MBSE approach envisioned to cover the whole life cycle of a satellite, starting from its initial conception to the management of the operational phase.

A cornerstone for ensuring modularity, reusability, and a high level of semantic precision is the notion of a Conceptual Data Model (CDM). A CDM is a meta-model providing the language for capturing and defining a specific aspect in the satellite model. In contrast to generic modeling languages such as SysML or UML, a CDM may be specific to a certain phase or a certain engineering discipline. In the technical memorandum ECSS-E-TM-10-23 [73] provided by the European Cooperation for Space Standardization (ECSS), a CDM is defined as a

“data model that captures the end-user needs in the end-user terms.”

Virtual Satellite provides a generic systems engineering language in which a CDM capturing one specific engineering aspect can be described. Here, we discuss such a CDM for the FDIR domain, developed for VirSat FDIR.

The workflow for performing studies with VirSat starts with a minimal set of concepts in the early phase design and then adds new CDMs to the data model

as the design matures and additional engineering aspects need to be considered. For example, in the context of system and FDIR design, the modeling would start by employing a CDM for modeling system elements, and then at a later stage, add the FDIR CDM into the model. Fig. 7.1 on the previous page illustrates how the model grows during the design phases.

The high level of specialization enables CDMs to be semantically precise and restricts models to avoid creating ones with an unclear interpretation. An example for a CDM that is actively being employed in the CEF is described in [74]. This CDM is used for creating Phase 0/A satellite models.

7.2 Generic Systems Engineering Language

In VirSat, CDMs are described using its Generic Systems Engineering Language (GSEL) [75]. The GSEL is partitioned into two types of elements: StructuralElements and Categories.

- **StructuralElements** are used to describe system decompositions into its various subsystems and parts, as well as relations between parts. An example of a relationship between StructuralElements is a product in a product list providing a product type to the actual instantiation in the satellite model.
- **Categories**, on the other hand, are used for tagging parts with the actual data information. Examples for attachable Categories are mass values, power consumption, interfaces, or – relevant for this work – FDIR information.

From the CDM definition of StructuralElements and Categories, the GSEL toolset generates a data model, and a UI for editing said data model within a VirSat application. CDMs are bundled by VirSat extensions. A VirSat extension is a VirSat application equipped with a set of CDMs and additional custom logic for processing the data of the CDM. Fig. 7.2 on the next page depicts the architecture of different VirSat extensions operating on a shared repository.

Extensions may share common concepts or be completely independent. When an extension accesses a repository, its concepts are stored in the repository alongside the satellite data model. This enables different VirSat extensions equipped with different sets of CDMs to communicate with each other.

To enable concurrent engineering, each instance of a StructuralElement is tagged with an owner. Only the owner can edit this instance and assign

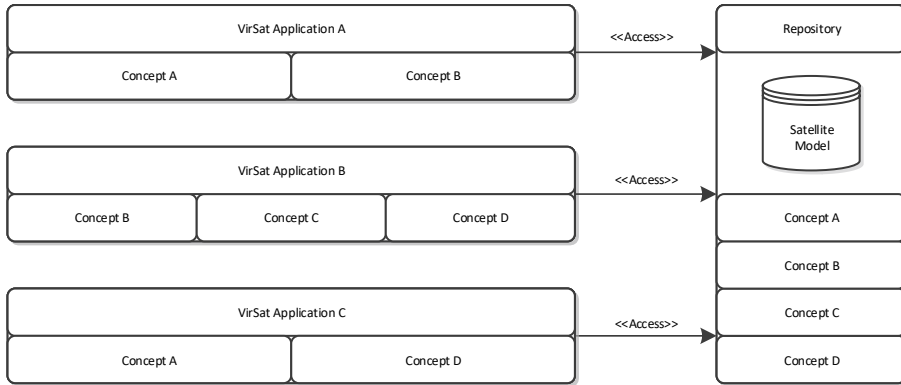


Fig. 7.2. Virtual Satellite 4 architecture with different VirSat extensions operating on the same repository.

Categories to it. In this manner, merge conflicts for storing the data models in repositories such as SVN or Git are avoided.

7.3 Virtual Satellite 4 FDIR

VirSat FDIR is a VirSat extension that provides modeling elements to annotate system elements with fault information as described in an FDIR CDM. As a Virtual Satellite extension, it inherits the tool capabilities from VirSat.

The tool aims to support FDIR engineering throughout all the phases of spacecraft design, starting from the design of redundancy concepts to the later design of onboard software FDIR. The implementation supports direct graphical modeling of DFTs/NdDFTs and recovery automata to create input fault trees. As another input language, VirSat FDIR employs the Galileo file format [76], which can also be used to describe NdDFTs as they are syntactically equal to DFTs. The file format has been slightly extended to incorporate *t*-MONITOR and *x*-RDEP gates.

The Galileo language has been implemented using XText [77]. All reliability data sets, state space sizes, benchmark times, and synthesized recovery automata in the evaluation performed in Chapter 8 have been generated using VirSat FDIR. The following gives some further details on the techniques applied in VirSat FDIR and their implementation.

The conceptual data model closely follows the objects that have been formally defined. Similar to the depiction convention for recovery automata, ϵ -loop transitions are not explicitly modeled. If some input E is undefined for some state, then it is assumed to be an ϵ -loop.

By default, VirSat FDIR employs standard DFT semantics from which the usual metrics such as MTTF, minimum cut sets, and so on are computed and saved in model artifacts dedicated to analysis results. When a recovery automaton is defined, VirSat FDIR employs the NdDFT semantics. Likewise, if no MONITOR gates are defined, the fully observable semantics is chosen, and as soon as at least one MONITOR gate is a part of the model, it is interpreted using PO semantics. For the PO case, additional custom-defined metrics have been implemented, namely:

- **Observability after time t :** The observability describes the ability of the failure of a node to be observed after time span t has passed. In terms of the BMA semantics, this metric corresponds to the probability of being in a state where the node has failed, and the failure has been taken into the observation set obs . We denote this probability with $\Omega(t)$.
- **Mean Time To Observation:** The Mean Time To Observation (MTTO) describes the long-term expected time it takes on average for a failed node to be observed by a monitor. To define the meaning of MTTO in terms of the BMA semantics, we also define a help metric, Mean Time To Unobserved Failure (MTTUF). The MTTUF describes the expected time for a node to fail without the failure being observed, or, in other words, the expected time to reach a state where the node has failed but is not in obs . For the MTTO, we then have $MTTO := MTTF - MTTUF$.
- **Steady State Observability:** The Steady State Observability (SSO) describes the long-run capability to observe a given node. It corresponds to the converged value of $\Omega(t)$ for $t \rightarrow \infty$.

The key focus of VirSat FDIR considered here is the synthesis algorithm. As optimization objectives, the user can pick any long-run metrics MTTF, SSA, SSO, and MTTO.

Since the CDM is conceptualized with the GSEL, VirSat FDIR can annotate any Virtual Satellite study with fault and recovery information without requiring domain-specific knowledge about the models that are being annotated. This especially includes very early Phase A models only consisting of a breakdown hierarchy of the system into its subsystems and equipment, and hence, during

the initial planning of the equipment redundancies. In later phases, software fragments can be tagged with faults to include software mechanisms into the failure model.

7.3.1 FDIR Conceptual Data Model

Our Conceptual Data Model for the FDIR domain deals with mainly three FDIR aspects: Modeling faults, detection, and recovery from them. There are no special modeling elements for isolation in the FDIR CDM, although one could consider the failed, probabilistic recovery actions of the PORAs as isolation actions trying to gain more information about the system. Moreover, since, in our framework, detection is modeled using an additional gate type in the fault tree, there are no special considerations for detection modeling in the FDIR CDM beyond introducing the MONITOR gate and metrics for the analysis. Overall, the FDIR concept itself can be divided into the following sub-concepts:

- **The Fault CDM**, which focuses on modeling NdDFTs and DFTs.
- **The Recovery CDM**, which focuses on modeling recovery automata.
- **The Analysis CDM**, which focuses on modeling analysis information, such as reliability curves, MTTF, and so on.

The CDMs are independent of the concrete structural decomposition of the system and only contains Categories. The actual system decomposition in terms of StructuralElements is expected to be defined in a separate concept engineered towards the domain objects that should be studied. In the following, the word *component* is used to refer to any element of such a structural decomposition. To provide out-of-the-box modeling capabilities, VirSat FDIR is equipped with the FDIR CDM and a default concept for modeling the system decomposition. The default structural decomposition, its application, and how it can be used to construct reusable, large-scale fault trees is briefly discussed in Section 7.3.3.

7.3.1.1 Fault CDM

The core element of the Fault CDM is the Fault category. It can be assigned to any component. To model the cause of a Fault, a meta-model following DFTs is employed. The BasicEvent category models direct causes of a Fault and is supplied with a failure rate and, optionally, with a repair rate. Additionally, the category can contain special repair rates for defined observation sets. For

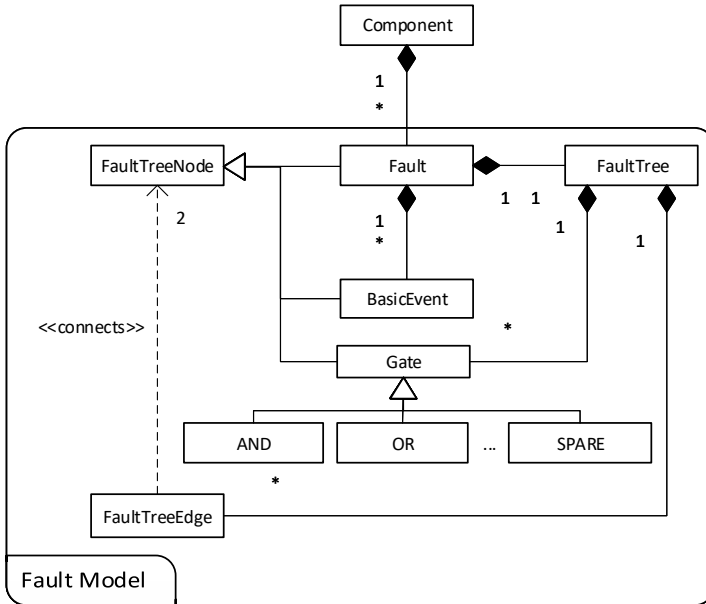


Fig. 7.3. Section of the FDIR CDM for modeling faults.

indirect causes, every Fault is also equipped with an FT, the Fault being the root of the FT. Every FT contains its local graph data, i.e., its edges and the gates describing the propagation from the lower level faults to the root failures. Fig. 7.3 summarizes the Fault CDM and illustrates the relations between the Categories.

To support the provision of fault information from component off-the-shelf (COTS) suppliers, the software supports a mechanism for importing fault trees described in the Galileo format. Likewise, the textual format is also used to export the FT model into an input representation for other external FT analyzer tools, such as STORM [78] or DFTCalc [79]. The latter in return provides various interfaces for other solver back ends. For high-performance analysis of large-scale fault trees using the deterministic standard DFT semantics, the user can set VirSat FDIR to employ STORM as the verification back end directly. STORM needs to be installed separately from VirSat FDIR.

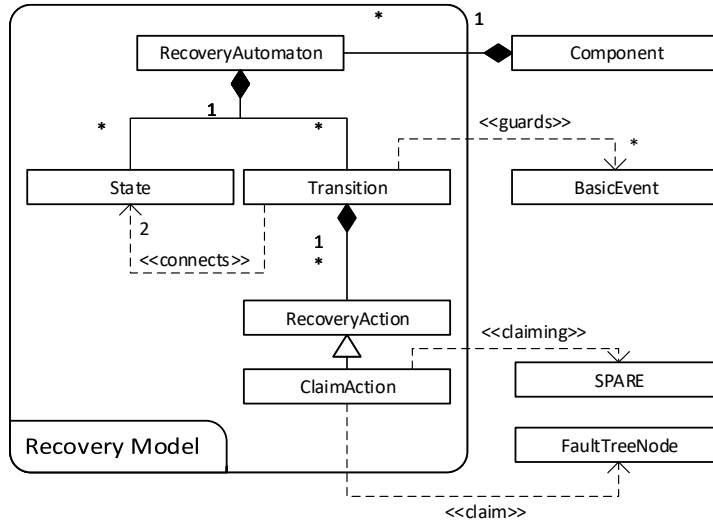


Fig. 7.4. The CDM section for modelling recovery.

7.3.1.2 Recovery CDM

For modeling the recovery aspect, we introduce, in this section, the model fragments that correspond to the recovery automaton from the theory. The model is straightforward, containing a classical automaton model with states and transitions. Each transition then has references to its start state, end state, and guards in the form of references to fault tree nodes. Each transition also contains the triggered recovery actions. Additional information such as optional timeout data for timeout transitions in PORAs may also be contained here. The overall Recovery CDM and its relation to the elements of other CDMs are illustrated in Fig. 7.4.

Since CDMs in VirSat are extensible, extensions of the FDIR CDM can provide additional recovery actions. To support this process, the Recovery CDM provides a generic RecoveryAction category from which recovery actions can derive. Here, these recovery actions cover the Claim and the Free action. On the implementation level, the semantics of these additional recovery actions can be injected into the semantics used by the state space generator.

7.3.2 Analysis CDM

Analysis results such as reliability, availability, observability curves, and other metrics are stored in dedicated categories. Each of these categories bundles a set of metrics. Metrics pertaining to a single Fault can be attached to the corresponding Fault. The following analysis categories have been defined:

- **ReliabilityAnalysis:** This category stores the MTTF and a reliability curve for a designated mission duration.
- **AvailabilityAnalysis:** This category stores the SSA and an availability curve for a designated mission duration.
- **ObservabilityAnalysis:** This category stores the SSO and MTTO, as well as an observability curve for a designated mission duration.
- **FMECA:** This category stores an FMECA generated from the fault tree model.
- **MCSAnalysis:** This category stores all the minimum cut sets up to a given maximum MCS size. For each MCS, we also store the MTTF, SSO, MMTO to classify the occurrence and observation probability of each MCS. Also computed in this analysis is the overall fault tolerance: The size of the smallest MCS.

7.3.3 Configuration Control

Not all subsystems need to be designed from scratch when designing a new system. Going one level deeper: For hardware components, especially off-the-shelf products with a high degree of reuse, the manufacturer may already have a prior fault tree analysis. For space systems, the satellite bus may require tailoring towards each payload, but even then, there may be previous fault tree artifacts that can be reused.

The idea of configuration control is to introduce a systematic way to reuse MBSE artifacts from manufacturers, previous studies, and so on, in a new study. The configuration control implemented in VirSat is delivered together with the Product Structure CDM. We consider here a simplified version of the Product Structure CDM defined in VirSat. The CDM is depicted in Fig. 7.5 on the next page.

This CDM describes a way to decompose the system in a manner that allows information to be reused. The CDM exploits VirSat's inheritance mechanism,

allowing structural elements to define run-time inheritance links. A structural element that inherits from another structural element obtains a copy of all its assigned categories, with the ability to further extend or override them.

A ProductTree (PT) represents a container for product definitions, called the ElementDefinitions (ED). An ED abstractly represents a component (Product). A ConfigurationTree (CT) represents a concrete instance of the system, for example, the satellite model, and contains ElementConfigurations (EC). ECs can be typed by EDs and inherit their assigned categories. This allows defining a run-time type-instance pattern (similar to classes and objects), where products can be defined independently of the mission, and then concrete instances of these products can be used to build the MBSE model of the system under study. In particular relevant here: Fault categories can be assigned to EDs. An example of the overall fault tree composition using the Product Structure CDM is given in Fig. 7.6 on the following page.

A single ED types two ECs, each inheriting the fault tree attached to the ED. The CT has a TLE defined. The overall system fault tree is created by composing the inherited fault trees. In this manner, product level FTA can be reused over multiple ECs and also over multiple missions, each with their own CT. Reusing the fault trees on the component level enables the user to compose large-scale fault trees from existing smaller fault trees. While we have mainly discussed hardware fault trees, this also applies to possible software fault trees.

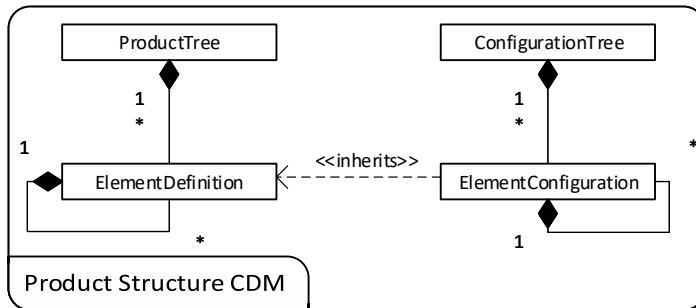


Fig. 7.5. Simplified Product Structures CDM.

7.3.4 Software Workflow for Synthesis

We briefly describe in this section how the synthesis procedure is integrated into VirSat FDIR. A special category *RecoveryAutomatonGen* describes the synthesizer configuration. The first Fault will be identified as the TLE, for which the automaton shall be synthesized by attaching it to a structural element. A property *objectiveMetric* allows setting the optimization metric. Eligible metrics are all long-run metrics MTTF, SSA, SSO, and MTTO. By default, VirSat FDIR is deployed with a native scheduler engine using value iteration for computing the optimal scheduler. Alternatively, the user can also choose the STORM engine as the verification engine.

7.4 Implementation Details

In order to efficiently implement the described DFT semantics, a series of techniques have been implemented. In the following, the main techniques are discussed.

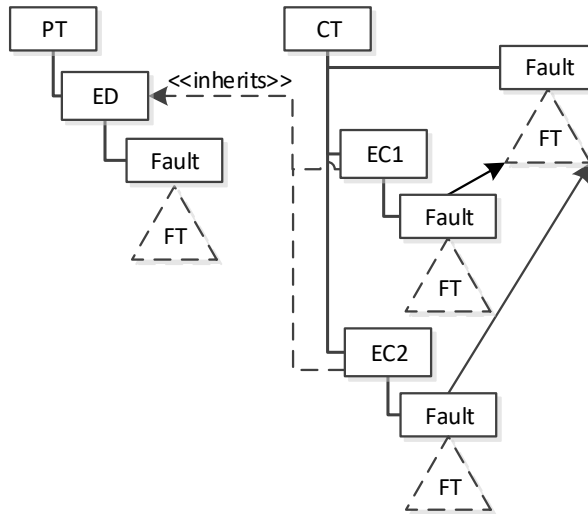


Fig. 7.6. Example of a Product Tree on the left side, and a derived Configuration Tree on the right side.

7.4.1 Preprocessing

In order to simplify the handling of the diverse landscape of fault tree gates, fault trees are simplified by replacing gates that are expressible through other gates. For instance, AND and OR gates are replaced by corresponding k -VOTE gates, and POR gates express all other priority gates. With this approach, the set of gates that need to be considered is reduced to k -VOTE, SPARE, FDEP, x -RDEP, and POR.

7.4.2 Representation of DFT states

Formally a DFT state is represented using a history of occurred BE sets. However, this leads to states with different occurrence orders of BEs to be different states. While this is desirable for various gates sensitive to the order of occurrence, such as priority gates or FDEPs, many gates do not require this information.

For each state, we therefore only save the order-sensitive BE sets in a history. For the other basic events, the failure information is stored in two bit-sets *failed* and *permanent*:

- *failed* represents the already discussed named function $failed(s)$. A node is in *failed* iff it is failed in the current state.
- *permanent* represents whether the state of a node can be changed. A non-permanent node can change its state either by failing or by being repaired.

We introduce an entry for every node in the fault tree in *failed* and *permanent*. Similar to the propagation of failure between fault tree nodes, the implementation also propagates the permanence of a node. If sufficient failure inputs for a node are set to permanent to determine that the output of a node is also permanent, then the node is also set to permanently failed. A basic event b is set to permanently failed if it fails and has a repair rate $r(b) = 0$. For static gates such as AND and OR, the AND gate is set to permanently failed iff all inputs are permanently failed, and the OR gate is set to permanently failed iff at least one input is set to permanently failed. In the case of the SPARE gate, if the primary is permanently failed and all spares are permanently failed, then also the SPARE gate is set to permanently failed. We exploit the permanence information in Section 7.4.3 to reduce the state space.

7.4.3 Canonical States

The direct application of the presented Markov automaton semantics to NdDFTs potentially yields vast state spaces. To keep the state space in a manageable size, we have applied a technique we refer to in the following as *canonical states*. When a basic event is set to failed, we also let all other basic events, that do not alter the future failure behavior of the fault tree, fail. This is primarily accomplished by exploiting the *permanence* bit-set. The permanence of a parent node can be backward propagated to all child nodes that only have outputs to permanent nodes. All permanent nodes, particularly basic events, can be safely set to failed.

For example, consider a sub-tree with an OR node and a set of basic events as children, of which one has permanently failed. Then, we can propagate and also set the OR node to permanently failed. Backward propagation then sets all other basic events to also be permanently failed. Since the OR node has already failed, further failures of the contained BEs do not affect the failure behavior of the fault tree. This way, we do not need to distinguish between the states where the basic events are operational or failed.

Note that if all parents of a basic event are permanently failed, we can set it to permanently failed even if the basic event is repairable. A Markov automaton state in which the maximum number of such BEs have failed is called a *canonical* state. We transform all states upon their generation into their respective canonical form.

7.4.4 Optimization Workflow

Performing the orthogonal merge requires the computation of the disabled inputs on all states. As the initial recovery automaton, which is extracted from a Markov automaton, can be very big, computing the set of disabled inputs for every state requires the computation of many set intersections. In addition to directly optimizing a recovery automaton with the Recovery Equivalence relation, we therefore also consider an optimization strategy where the recovery automaton is first optimized using classical trace equivalence \approx (Def. 5.3). The resulting recovery automaton with reduced state space is further optimized using the \approx_R relation (Def. 5.5). In addition, the FAIL rule is applied in the beginning for both workflows as merged FAIL states may reveal new orthogonally equivalent states. The opposite has not been observed, but also not been disproven. The resulting optimization workflows are visualized in Fig. 7.7 on the next page. We

later empirically verify in our evaluation that the refined workflow from Strategy #2 is indeed an improvement.

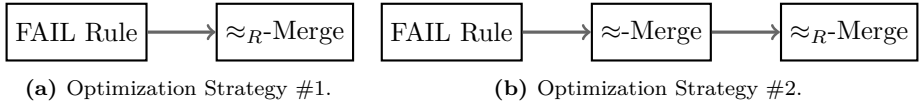


Fig. 7.7. Optimization workflows.

7.4.5 Reducing the Number of Timeout States

PO states generate a series of timeout transitions in the RA. However, not all of them have a different recovery behavior. Therefore, we apply a simple approach to reduce the number of timeout states by simply merging bisimilar RA states and adding the total timeouts on the timeout transitions. In the RA-MA synchronization construction, we then skip over τ -transitions until the total mean time to happen equals the accumulated timeout on the RA transition.

7.4.6 Selecting Optimal Transitions on the Fly

In the semantics, we explore the full state space for every possible non-deterministic decision and afterward choose the decision based on the optimal scheduler. However, if we can already conclude during run-time that from a set of certain decisions, we can identify that some of them are definitely not viable compared to the other decisions, then we could discard these non-deterministic state successors without exploring them further. In the following, we refer to such a decision during the state space generation as a run-time optimal transition selection.

The key idea is to identify decisions that lead directly into a fail state. Even if we do not know the full state space and thus usually cannot perform an early metric evaluation, in the case of reaching a fail state within one decision, we can safely choose non-fail states over the fail states. This avoids state space generation of empty recovery actions and FREE actions that the optimized scheduler will never choose as they will directly lead to a failure.

In the PO case, we apply a slight modification of this idea: Since the fail property in PODFT states is a fuzzy probability, non-deterministic states may have a high number of successor states with a non-zero fail state probability. The optimal transition selection here chooses the state with the least failure

7.4. IMPLEMENTATION DETAILS

probability. If there are few states with a non-fail state probability of 1, the number of viable successor states decreases drastically.

Chapter 8

Evaluation

In this chapter, we focus on evaluating the NdDFT methodology. Since the approach is state-based, this raises the immediate question regarding its scalability and how the introduction of non-determinism contributes to the state space explosion. Some measures, in particular modularization, to counteract said state space explosion have been proposed. However, we have yet to verify the effectiveness on practical, real-life data sets. In the following, we will first present a general experimental setup that we will be reusing for all following experiments. We have chosen the Fault tree FOResT (FFORT) benchmark set, which is a compilation of major literature fault tree case studies, as a basis to perform our evaluation. We first present some more details on this benchmark set and elaborate on the additional data sets that have been compiled into it. Afterwards, we will evaluate the scalability of various configurations of NdDFTs (Repairable, partially observable, and so on). Furthermore, we will also perform some experiments to investigate the effectiveness of our presented recovery-equivalence-based reduction approach.

8.1 Experiment Setup

The FFORT benchmark set introduced in [80] was used as a source of fault tree benchmarks to evaluate our proposed techniques. FFORT is an online fault tree database with fault trees collected from scientific literature for the primary purpose of benchmarking. We have selected fault tree families from the FFORT

benchmark set that contain at least one SPARE gate but do not employ the authors' custom fault tree extension of inspection modules (IM). Therefore, we can guarantee that all experiments contain some non-determinism. The following fault tree families from the FFORT benchmark fulfilled the selection criteria. (The graphic symbols refer to the evaluation charts shown in the subsequent sections)

- **Active Heat Rejection System (AHRs ◊)**. The AHRs is made up of thermal rejection units of which only one is needed for the system to function.
- **Cardiac Assist System (CAS ●)**. The CAS models a hypothetical cardiac assist system with redundant CPUs, motors, and pumps.
- **Electro-Mechanical Actuator (EM _)**. The model focuses on common-cause failures in an electro-mechanical actuator.
- **Hypothetical Example Computer System (HECS □)**. The HECS fault trees model computer systems including their processors, memory modules, buses, consoles, operators, and software.
- **Hypothetical Example Multi-Phase System (HEMPS ■)**. The HEMPS model is a demonstrator of a system designed for a multi-phase mission.
- **Mission Avionics System (MAS *)**. The MAS models represent mission- and safety-critical systems with high redundancy. Components include hardware, software and vehicle control subsystems, and system management.
- **Multiprocessor Computing System (MCS ×)**. The MCS model computers with power supplies, memory modules, hard disks, and connecting buses. The benchmarks have been enriched with instances from [62].
- **Nuclear Power Plant Water Pumping System (NPPW ⊗)**. The model represents a nuclear power plant system.
- **Railway Crossing (RC ◊)**. The RC fault tree collection models level railway crossings with sensors, motors, and controllers. The models come in two variations (sc and hc), representing the controller being a single basic event or hypothetical example computer system, respectively.

- **Vehicle Guidance System (VGS_Δ)**. The VGS models are industrial case studies dealing with variants of safety concepts for vehicle guidance systems.

The benchmarks were carried out with an Intel(R) Xeon(R) W-2155 CPU, 16GB of RAM, and a timeout of 600s (10min). Note that this hardware is not identical to the hardware used to produce the prior results published in [8]. In particular, the accessible RAM has been increased from 4GB to 16GB. The software, the experiment setup, all experiments, and all results can be found at [72].

The number of solved instances, the number of timeouts, the number of out-of-memories (OOMs), and the total solving time were logged for each of the following configurations. *Solving an instance* here refers to successfully synthesizing a recovery automaton given an input NdDFT. There is an exception in the subsection focusing on orthogonal state-space minimization. There, *solving an instance* refers to computing a minimized recovery automaton given a synthesized recovery automaton. For each experiment configuration, we give a tabular summary of the results and then have a closer look at the growth behavior in relation to the number of fault tree nodes.

8.2 Fully Observable Scalability Experiments

This section focuses on evaluating configurations for the fully observable case. We investigate the baseline scalability of our approach without applying modularization. From there, we validate the effectiveness of the modularization approach by comparing them to each other. Moreover, we check how repair impacts scalability. Since the FTs from the FFORT benchmarks are not repairable, we first define a rule for generating repairable NdDFTs. We apply the simple rule for attaching a repair rate to every basic event, with the repair rate equal to the failure rate. In total, the setup gives us the following four configurations covering the combinations of using repair semantics and modularization:

- NdDFT without modularization
- NdDFT with modularization
- Repairable NdDFT without modularization
- Repairable NdDFT with modularization

8.2.1 NdDFT Experiments

We consider in this subsection the non-repairable NdDFT configurations, on the one hand with modularization, and on the other hand without modularization. A summary of the results is given in Tab. 8.1.

Table 8.1: Summary of benchmark results: NdDFT without repair.

modularization	solved	total	timeouts	ooms	solveTime [s]
no	21	156	121	14	936
yes	143	156	13	0	602

As hypothesized prior, not applying modularization leads to state-space explosion, causing many cases of TOs and OOMs and only solving a minor amount of 21 instances. It is safe to conclude that the NdDFT approach does not scale without modularization. However, by applying the modularized approach, we synthesized RAs for all instances but 13. The experiments validate the overall scalability of RA synthesis for NdDFTs. The following charts give a closer look at the results of the experiments.

Fig. 8.1 shows a detailed time comparison between the synthesizer with and without a modularizer, respectively. The dashed line marks where both algorithms require equal time. Timeouts and out-of-memory results have been placed on the outer lines and are labeled with TO and OOM, respectively.

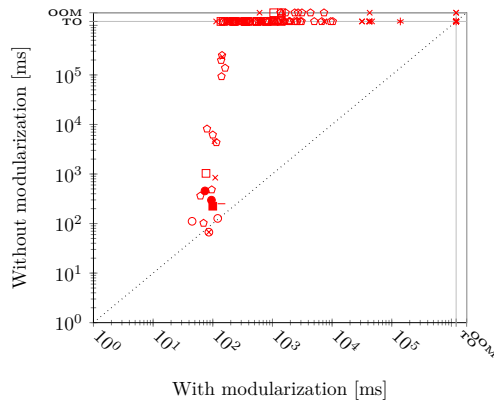


Fig. 8.1. Modularization vs no modularization.

8.2. FULLY OBSERVABLE SCALABILITY EXPERIMENTS

The chart reinforces the observations we have made from the summary. The approach without modularization quickly heads towards TOs. On the other hand, the modularized approach mostly scales. The families which not even the modularized approach can handle are primarily MAS_{\times} and MCS_{\times} . Both of these feature instances with large modules and a high number of SPARE gates due to a high degree of spare sharing. On the other hand, the cases which can be solved without modularization are mainly those with a small number (<5) of SPARE gates. We will see in the following figures that a primary driver for whether an instance can be solved or not is the number of SPARE gates. Fig. 8.2 shows the time measurement breakdown compared to the number of SPARE gates in the input instance.

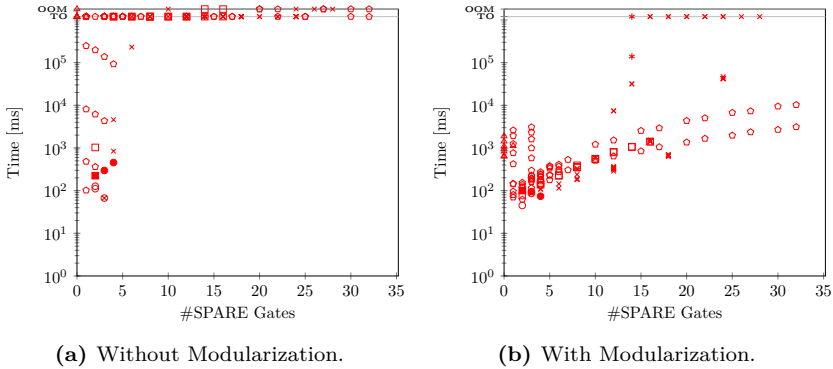


Fig. 8.2. Time measurement breakdown compared to the number of SPARE gates.

We can observe that the unmodularized approach cannot handle a growing number of SPARE gates. For the modularized approach, we can also see that for a small number of SPARE gates, we can synthesize RAs within the timeout. At >15 SPARE gates, the MC_{\times} instances break out and trend towards a TO. However, we can also see that other fault tree families, here primarily the RC_{\circ} instances, can still be solved even for a large number of SPARE gates. A special remark should also be given on the VGS_{Δ} instances with no SPARE gates. It appears that from the VGS family, only one instance has a SPARE gate, while the others do not. In the modularized case, this means that the entire tree of these SPARE gateless instances is simply discarded in the trimming step.

We conclude this evaluation by examining how the modularized approach scales with an increasing number of nodes. Fig. 8.3 on the next page shows how the synthesizer scales in terms of total nodes in a fault tree.

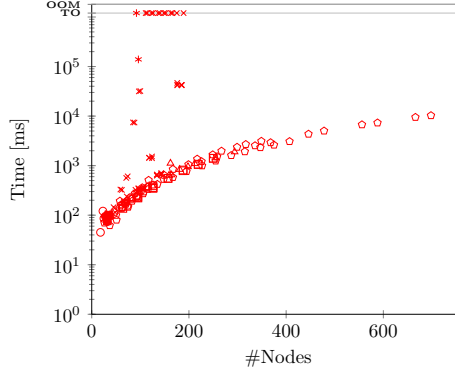


Fig. 8.3. Time measurement break-down for growing number of nodes.

The figure shows that, in general, the number of nodes does not cause an exponential blow-up in the synthesis algorithm. However, we can also again observe the breaking out MC_x instances. Moreover, they already start breaking out for a small number of nodes. Overall, we can infer from both observations that instances with a high density of SPARE gates are problematic for the synthesis algorithm.

8.2.2 Repairable NdDFT Experiments

We consider here configurations using repairable NdDFTs. As before, we consider the two cases of employing modularization and not employing. To create the benchmark from the FFORT benchmark set, for every fault tree, we have created a repairable version. Each basic event is also equipped with a repair rate equal to its failure rate in this repairable version. A summary of the results is given in Tab. 8.2.

Table 8.2: Summary of benchmark results: NdDFT with repair.

modularization	solved	total	timeouts	ooms	solveTime [s]
no	6	156	123	27	549
yes	82	156	19	55	1651

Not employing modularization, we were only able to solve six instances. Considering our prior observations regarding the scalability issues on regular

8.2. FULLY OBSERVABLE SCALABILITY EXPERIMENTS

NdDFTs, it is not surprising to see the synthesis algorithm without modularization fairing worse on the harder, repairable instances. We can also see a reduction of solvable instances when employing modularization. Going further, despite the decrease of solved instances, we can also observe an increase of the total solve time by a factor of 2.75. Overall, while we can solve over half of the instances, we can conclude that RA synthesis on repairable NdDFTs is problematic. In the event of triggering a TO for analyzing a repairable NdDFT, it should be considered if the fault behavior can be meaningfully modeled with less repair rates.

We can also observe that the number of OOMs increases significantly from 0 to 55. The data reveals that these OOMs do not occur exclusively in the construction of the MA. 30 of the 55 OOMs occur during the RA composition for the whole NdDFT. We visualize the blow-up in the RA state space in Fig. 8.4 by comparing the RA size post-composition from the non-repairable case with the repairable case.

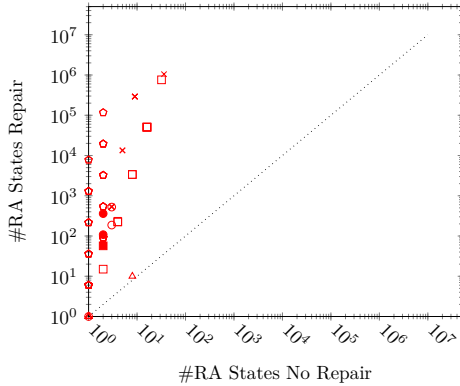


Fig. 8.4. Increase of RA states for repair vs no repair cases.

We can observe that the state space increases massively up to 6 orders of magnitude. The increase may be due to the lack of orthogonal state space reduction in the repairable RA, which is not applicable for this case. It may also be due to the RA strategies becoming intrinsically more complex.

In the following, we take a closer look as to which instances are problematic. For this, we first consider a detailed breakdown between the configurations in Fig. 8.5 on the next page again.

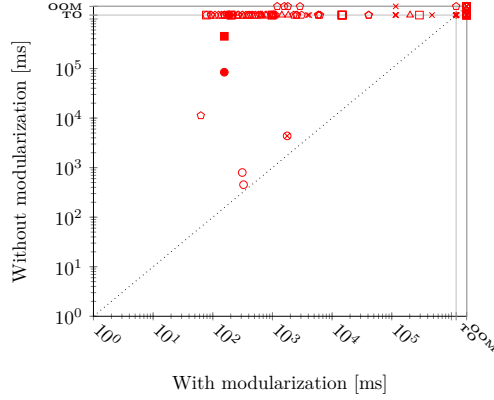


Fig. 8.5. Modularization vs no modularization (Repair).

Since most instances for the non-modularized case run into a TO or OOM, we only have a few instances left in the plot. The plot shows us the few instances we could solve without modularization. In the non-modularized case, we were able to solve instances of $AHRS_{\circ}$, $NPPW_{\otimes}$, RC_{\circ} , CAS_{\bullet} , and $HEMPS_{\blacksquare}$. All the solved instances share in common that they only have small fault trees with less than 20 nodes and only 4-5 BEs and 2-3 SPARE gates.

We revisit the time measurement breakdown against the number of SPARE gates in Fig. 8.6 on the following page to visualize better the scaling issues arising from having a large number of SPARE gates.

The figure shows a similar trend curve as observed before in the non-repairable experiments for the non-modularized case. However, we can also observe that a large number of SPARE gates (>10) make the instances intractable for the modularized case. Already for 15 SPARE gates, we can no longer solve even a single instance. Overall, we can conclude that if we want to consider instances with many repairable BEs, we need to be judicious with our usage of SPARE gates, or otherwise, the RA synthesis becomes unfeasible.

Finally, we consider how the synthesis scales overall with the number of nodes in Fig. 8.7 on the following page. Note that modularization is employed.

Qualitatively, we can observe similarities to the non-repairable curve. We can see a curve of experiments that scales well even with an increasing number of nodes. This curve mainly consists of the RC_{\circ} cases, and more specifically, by its sc variation, whose instances have a structure that is easy to modularize. The hc

8.2. FULLY OBSERVABLE SCALABILITY EXPERIMENTS

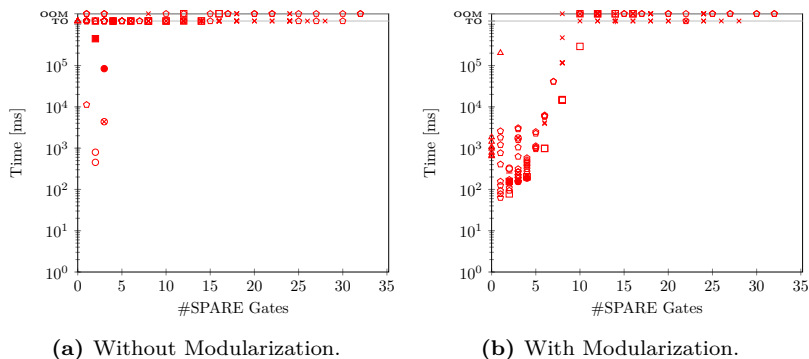


Fig. 8.6. Time measurement breakdown compared to the number of SPARE gates (Repair).

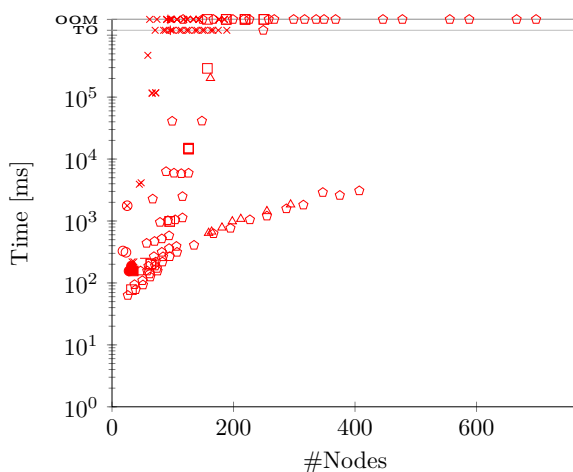


Fig. 8.7. Time measurement break-down for growing number of nodes (Repair).

variation, on the other hand, follows the other trend line that includes all other benchmark families and exponentially grows towards a TO. Overall, the results show that RA synthesis on repairable NdDFTs quickly becomes unfeasible when applied on instances where we cannot obtain small modules.

8.3 Recovery-Equivalence-Based State Space Reduction Experiments

This section evaluates the effectiveness of the recovery-equivalence-based state space reduction techniques. Since the RA minimization requires a synthesized RA, the benchmark set is further limited to those fault trees for which we could successfully synthesize an RA. For the experiments, we log two pieces of information: Firstly, we log the overall state space reduction using RA-equivalence, that is, using all RA state space reduction techniques presented in Section 5.3. Secondly, we also log the contribution of the trace-based equivalent state space reduction in the overall reduction process. The results are shown in Fig. 8.8.

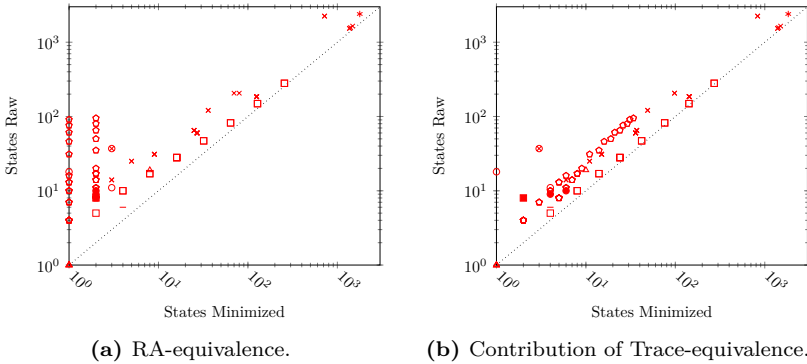


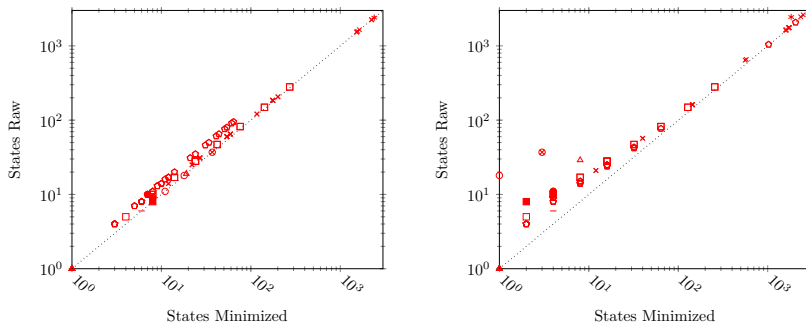
Fig. 8.8. RA state space reduction results. RA-equivalence vs Trace-equivalence.

The figure shows a significant reduction contribution from conventional trace-based equivalence. However, the overall effect is limited and does not reach an order of magnitude. We can infer that, in general, synthesized RAs are not well suited for trace-based state space reduction techniques. On the other hand, the RA-equivalence proves to be effective and, in some cases, even reduces the RA state space by two orders of magnitude. In fact, in many cases, we obtain minimal states where the RA only contains 1 or 2 states. Families on which the orthogonal RA state-space reduction is not adequate are primarily MCS_x and HECS_□. Investigating the commonality between these two reveals that these are both cases where SPARE gates have shared spares. RA reduction thus becomes more difficult as more states are required to represent the optimal

8.3. RECOVERY-EQUIVALENCE-BASED STATE SPACE REDUCTION EXPERIMENTS

strategy. Overall, we can learn from the data that RA-equivalence-based state-space reduction works well in eliminating RA blow-up resulting from states that execute trivial strategies, such as always claiming no matter what the state is. We only observe a slight improvement over the trace-based equivalence approach for complex strategies.

We have compared trace-based equivalence to the RA-equivalence-based approach, the latter including technically two rules: Orthogonal state-space reduction and the fail reduction rule. To evaluate the contribution of the fail reduction rule, we further evaluate the contribution of the fail state reduction rule. For this purpose, we consider, first of all, the direct contribution to the state space reduction as before. In addition to this, we also consider a configuration where the fail reduction rule is left out entirely to evaluate how it affects the overall workflow. The two figures in Fig. 8.9 visualize the measured results.



(a) Contribution of fail state reduction. (b) RA-equivalence without fail rule.

Fig. 8.9. Effects of the RA fail state reduction rule.

Fig. 8.9a logs the number of states removed by the fail reduction rule. From it, we can observe that the fail rule itself is not a major contributor to the overall RA state-space reduction workflow. However, we also consider an alternate configuration where we entirely turn off the fail state reduction rule. The results of the total state space reduction in this configuration are shown in Fig. 8.9b. First, we observe that the curve significantly differs from the results in Fig. 8.8 on the previous page. The entire RA minimization workflow is far less effective and cannot produce the many 1 or 2 state automata witnessed in a full minimization setup. In fact, due to the increased size, we could not synthesize as many RAs as we did before. In the setup with the disabled Fail reduction rule, we could only solve 126 instances; the other instances timed out during the RA composition

due to the large RAs in the individual modules. From these observations, we can conclude that while the rule itself does not contribute to a major state space reduction, it does help the orthogonal state-space reduction to identify more recovery equivalent states.

Next, we verify our proposition that the refined Strategy 2 from Fig. 7.7 on page 142 outperforms directly applying the recovery equivalence. For that purpose, we compare the run-time of the two strategies against each other. The results are shown in Fig. 8.10. For most instances, the strategies yield equal performance. However, for larger MC_{\star} and RC_{\circ} instances with also large recovery automata, we obtain a significant blow-up in run-time, yielding even many TO events.

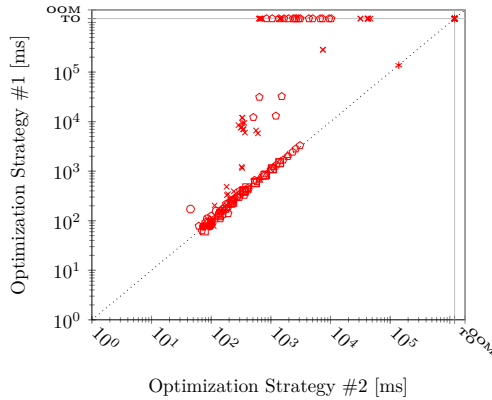


Fig. 8.10. Optimization Strategy #2 vs Optimization Strategy #1.

We conclude this section by considering a final experimental configuration: In our workflow, we only reduce the RA of each module and then perform the composition operation. Afterward, we do not perform an additional RA reduction. This raises the question of whether there are advantages of performing another state-space reduction step after the composition. The results of this configuration are illustrated in Fig. 8.11 on the following page. We have indeed found some cases where some additional states could be removed. Overall, however, we have not found a significant improvement.

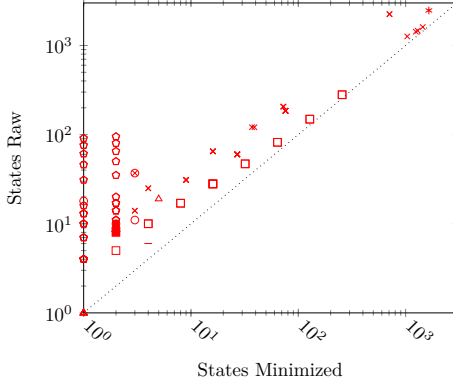


Fig. 8.11. Repeating RA minimization workflow after RA composition.

8.4 Partially Observable Scalability Experiments

In this section, we consider experiments to evaluate the PODFT semantics. We saw in the semantic definition of PODFTs that we needed to treat the case of immediate and delayed observers separately. Therefore, we further differentiate the following classes for the experiments:

- PODFTs only with immediate observers and without repair (PO)
- PODFTs with delayed observers and without repair (PO Delay)
- PODFTs only with immediate observers and with repair (PO Repair)
- PODFTs with delayed observers and with repair (PO Delay Repair)

We take the original FFORT benchmarks and add appropriate MONITOR gates to generate a benchmark set. However, the number of MONITOR gates influences state space construction, and it would be interesting to quantify this influence. We therefore define the auxiliary *observability level* parameter. We say that a PODFT has an observability level $i \geq 0$ iff all nodes up to depth i are observed by a MONITOR gate. In the case of $i = 0$, only the TLE is observed by a MONITOR gate. Using this parameter, we can investigate how scalability is impacted as we start with little information (only the root node is observable) and incrementally increase the observability level. Overall, we obtain the following constructions rules to derive PODFT benchmarks for a

given observability level i from the original FFORT benchmark set: For any node n with a depth of i or less, create a new MONITOR gate m and add an observation propagation from n to m . For delayed cases, add an observation delay of 1 time unit for any MONITOR gate on depth i . Note that all nodes on depth $i - 1$ are still immediately observable in the delayed case, and only the nodes on depth i are delayed observable. For the repair rates, we proceed as before and apply repair rates equal to the failure rates for every basic event. We compare in the following subsections with increasing observation levels:

- How the individual solve times evolve
- How the number of solved instances are impacted

We limit the scope of our investigations to the observation levels 0 to 5, the latter being the depth by which most FFORT benchmark set entries have placed their SPARE gates. We expect the results of observation level 5 to be close to the results of the fully observable case.

8.4.1 Configuration: PO

In this section, we consider the configuration of having no repair rates and all MONITOR gates having a delay of 0 time units. A summary of the results is given in Tab. 8.3. In Fig. 8.12 on the following page, we have visualized these results by plotting the observation level against the number of solvable instances.

Table 8.3: Summary of benchmark results: PO without delay.

observabilityLevel	solved	total	timeouts	ooms	solveTime [s]
0	11	156	125	20	16
1	40	156	81	35	320
2	99	156	57	0	150
3	104	156	52	0	403
4	126	156	24	6	649
5	142	156	14	0	730

First of all, the data shows that partial observability majorly impacts the tractability of the RA synthesis. On observability level 0, that is, only the root node is observable; barely any instances can be solved. On the other hand, at observability level 5, we can synthesize nearly as many RAs as we could in the fully observable case. As for the remaining unsolved instance, it is a case from

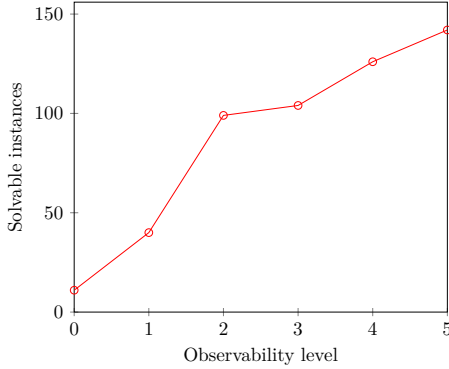
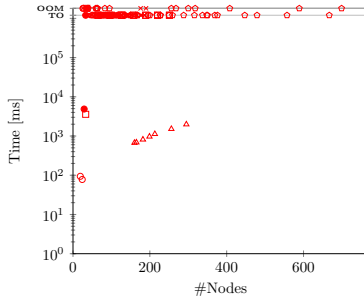


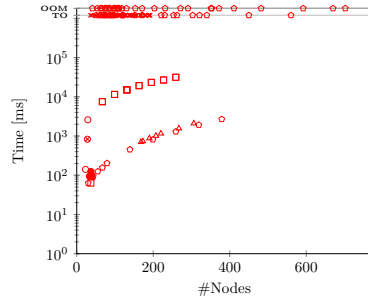
Fig. 8.12. Solvable instances for observability levels 0 to 5.

the VGS family with a SPARE gate and a fault tree depth of 7. In other words, in this case, the observability level 5 is not enough to fully observe the entire tree. For the evolution of the solved instance count between the observability levels 0 and 5, we can see a continuous increase in the number of solved instances. Especially from observation levels 1 to 2, we can see a major jump. A look into the benchmarks reveals that this jump is primarily due to an increase in the number of modules. That lower observability levels cause the modularization to find less modules is consistent with the additional modularization restrictions introduced in Section 6.4.4. Regarding the evolution of the solve times, there is an interesting anomaly occurring at observability level 2: The solve time with 150s is significantly low despite the already high count of solvable instances.

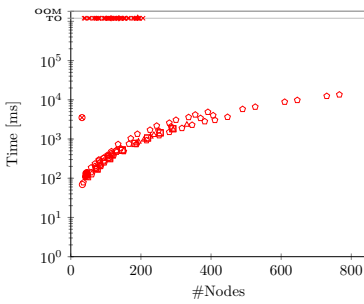
The plots given in Fig. 8.13 on the next page give a detailed breakdown of the solve time measurements for each observability level. We can see in Fig. 8.13a on the next page that the only initially solvable instances are from VGS_{Δ} and have no SPARE gates and an assortment of small instances. Fig. 8.13b on the next page shows that $HECS_{\square}$ and RC_{\circ} become solvable with observability level 1. As indicated from the tabular data, observability level 2 already resembles the fully observable results. Mostly the MC_{\star} family is currently not solvable. The MC_{\star} family becomes solvable over the course of the additional observability levels, as can be seen in the remaining figures until the final curve closely resembles the fully observable case from Fig. 8.3 on page 150.



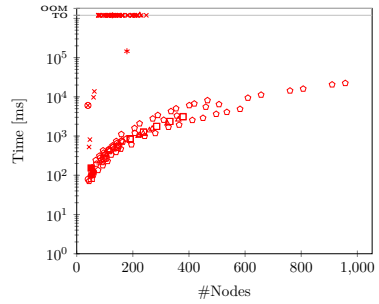
(a) Observability level 0.



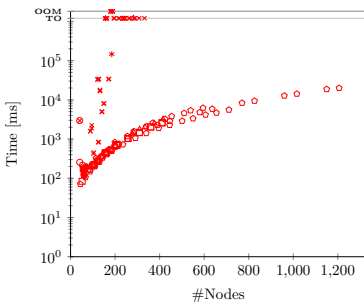
(b) Observability level 1.



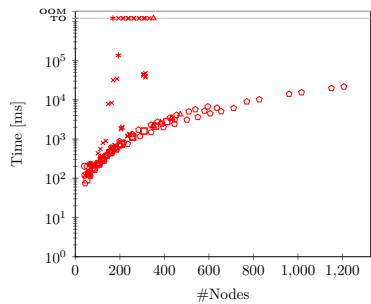
(c) Observability level 2.



(d) Observability level 3.



(e) Observability level 4.



(f) Observability level 5.

Fig. 8.13. Time measurement breakdown for observability levels 0-5.

8.4.2 Configuration: PO Delay

In this section, we consider the configuration of having no repair but a delay in the MONITOR gates. For a given observation level i , all MONITOR gates observing a node on depth i have an assigned delay of 1 time unit. A summary of the results is given in Tab. 8.4. In Fig. 8.14, we have visualized these results by plotting the observation level against the number of solved instances.

Table 8.4: Summary of benchmark results: PO with delay.

observabilityLevel	solved	total	timeouts	ooms	solveTime [s]
0	11	156	126	19	17
1	40	156	80	36	409
2	98	156	56	2	227
3	27	156	119	10	414
4	83	156	63	10	247
5	37	156	117	2	316

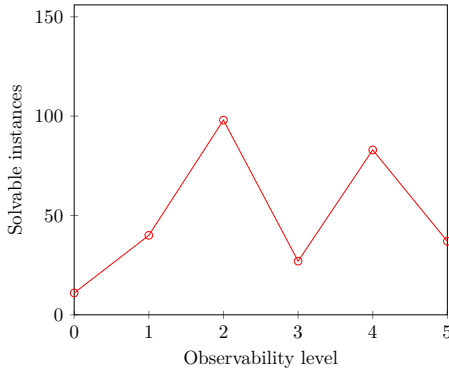


Fig. 8.14. Solvable instances for observability levels 0 to 5.

Note that the results for observability levels 0 to 2 are close to identical to the PO without delay case results. However, we can observe a substantial deviation from the PO case on the higher observability levels. Most importantly, the number of solved instances no longer increases continuously. Instead, we can observe two peaks at observability levels 2 and 4. Considering the complex state space construction rule we introduced for the delay case, this is not surprising.

The state space needs to encode both the possibility of observing an event and the possibility of being delayed. The data suggest that this construction rule leads to a significant negative impact to the point where we can solve only very few instances on observability level 5. Note that increasing the observability level also increases the number of immediate observers on the level $i - 1$ for observability level i . Due to these immediate MONITOR gates on the lower levels, the modularization algorithm can identify more modules and thus, in a sense, counters the negative impact of the delayed MONITOR gates.

The detailed breakdown for each observability level is given in the plots in Fig. 8.15 on the following page. Fig. 8.15a on the following page to Fig. 8.15c on the following page differ very little to their respective PO without delay counterparts, except some instances RC_{\diamond} and CAS_{\bullet} performing notably worse in the PO with delay case. On observability level 3, in Fig. 8.15d on the following page, we can see the drop of solvable instances: For a large number of RC_{\diamond} and $HECS_{\square}$ instances, we are suddenly no longer able to synthesize RAs due to the delayed MONITOR gate construction. On observability level 4, as Fig. 8.15e on the following page shows, the MC_{\star} instances become solvable due to achieving good modularization. Finally, for observability level 5, displayed in Fig. 8.15f on the following page, we can observe that only some easy MC_{\star} and the SPARE gate-less VGS_{Δ} instances remain solvable. We conclude overall that delayed MONITOR gates should be employed sparingly.

8.4. PARTIALLY OBSERVABLE SCALABILITY EXPERIMENTS

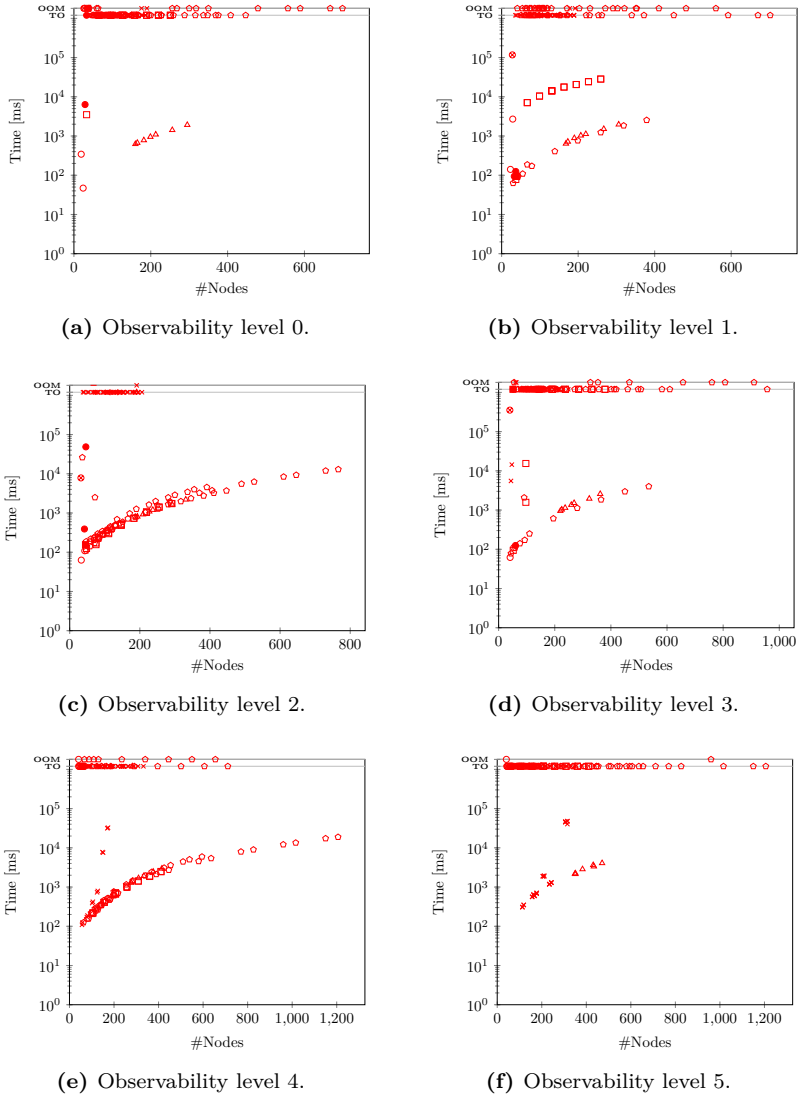


Fig. 8.15. Time measurement breakdown for observability levels 0-5.

8.4.3 Configuration: PO Repair

In this section, we consider the configuration of having repair rates and all MONITOR gates having a delay of 0 time units. For the repair rates we use the same setup as before and set them to the same value as the failure rates. A summary of the results is given in Tab. 8.5. In Fig. 8.16, we have visualized these results by plotting the observation level against the number of solvable instances.

Table 8.5: Summary of benchmark results: Repairable PO without delay.

observabilityLevel	solved	total	timeouts	ooms	solveTime [s]
0	9	156	134	13	22
1	22	156	132	2	25
2	77	156	59	20	666
3	72	156	52	32	767
4	77	156	32	47	1112
5	80	156	20	56	1584

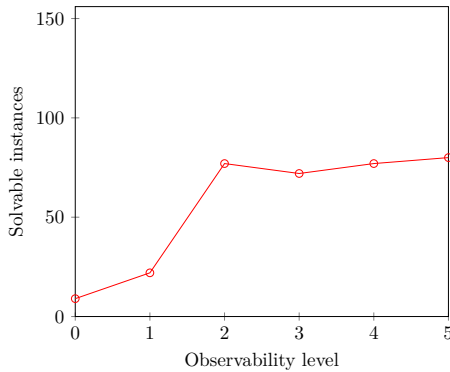


Fig. 8.16. Solvable instances for observability levels 0 to 5.

The results show that already on observability level 2, the number of solved instances closely approaches the fully observable case. Overall, adding partial observability to repair does not seem to have a major impact as adding it to the non-repair case. We can even find that the solve time for the observability level 2 is at only 666s despite only solving three instances less than observability level

5. We can only also observe a significant reduction of OOM events. Furthermore, after a peak at observability level 2, the number of solvable instances first decreases. To investigate these different results, we take a closer look at the RA sizes again. In Fig. 8.17, we compare the RA sizes of the PO Repair case on observability level 2 to the fully observable repair case.

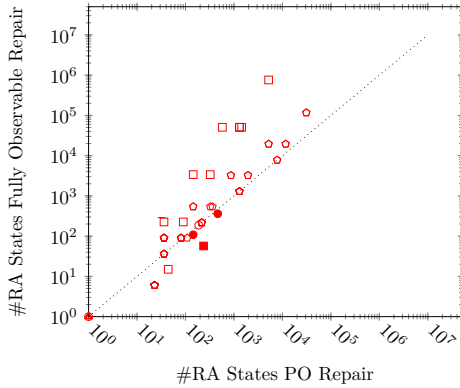
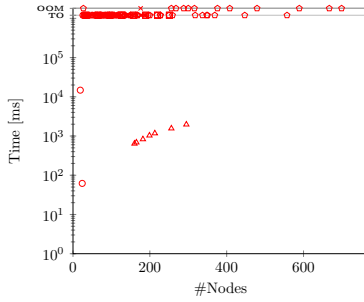


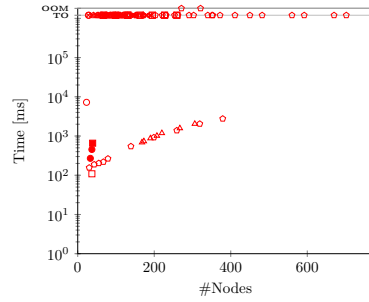
Fig. 8.17. Increase of RA states for partially observable repair vs fully observable repair case.

We have already confirmed that the RA size increases dramatically when going from the non-repairable to the repairable case. The figure shows that while the fully observable case for small instances tends to yield smaller RAs, the larger instances shift towards providing smaller RAs for the partially observable case. Since fewer events are observable, the RA also needs to process fewer inputs and consequently also does not need to memorize the inputs it cannot observe. We have not observed this significantly impacting the partially observable, non-repairable case. However, for the repairable case, reducing the number of observable events can significantly reduce the overall optimal RA size.

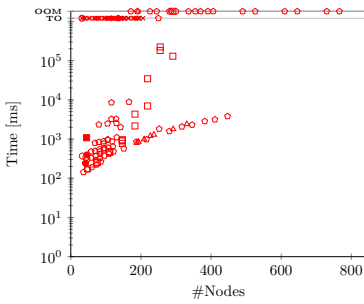
The detailed breakdown for each observability level is given in the plots in Fig. 8.18 on the next page. Note that from observability level 3, depicted in Fig. 8.18d on the next page, onwards, the curves are strikingly similar to the fully observable, repairable case from Fig. 8.7 on page 153. The differences between them are too insignificant to make any major observations. Overall, we conclude that adding partial observability to repairable instances does not lead to a significant blow-up, provided enough levels remain observable to perform modularization.



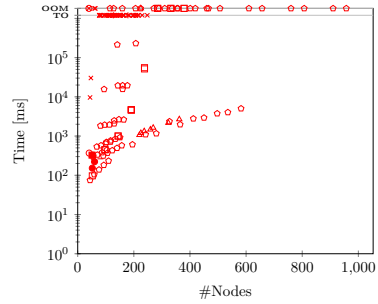
(a) Observability level 0.



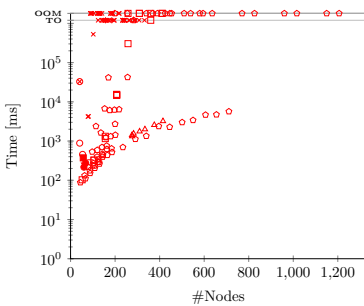
(b) Observability level 1.



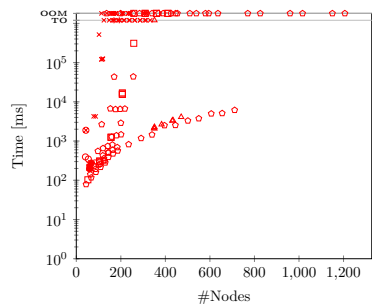
(c) Observability level 2.



(d) Observability level 3.



(e) Observability level 4.



(f) Observability level 5.

Fig. 8.18. Time measurement breakdown for observability levels 0-5.

8.4.4 Configuration: PO Delay Repair

In this section, we consider the configuration of having repair rates and delays in the MONITOR gates. As before, for a given observation level i , we assign all MONITOR gates observing a node on depth i a delay of 1 time unit. A summary of the results is given in Tab. 8.6. In Fig. 8.19, we have visualized these results by plotting the observation level against the number of solvable instances.

Table 8.6: Summary of benchmark results: Repairable PO with delay.

observabilityLevel	solved	total	timeouts	ooms	solveTime [s]
0	9	156	141	6	24
1	22	156	132	2	30
2	72	156	62	22	528
3	24	156	123	9	287
4	48	156	78	30	986
5	11	156	117	28	518

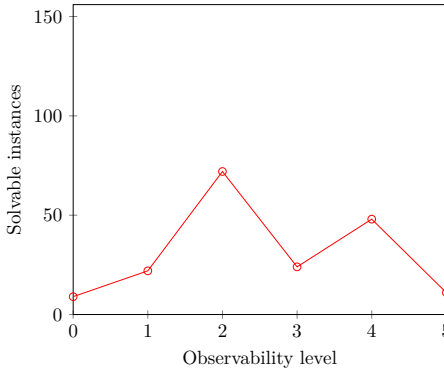


Fig. 8.19. Solvable instances for observability levels 0 to 5.

We can observe the same pattern from the PO Delay case. The data reveals two peaks on observability levels 2 and 4. Moreover, on observability level 5, barely any RAs can be synthesized. What stands out is that the peak on observability level 2 does not differ too greatly from the peak of the PO Repair case without delay. Only five instances less can be solved, despite the addition

of the delayed MONITOR gates. On all other observability levels, however, the performance of the PO Delayed Repair class does not scale.

Before we look at the detailed breakdown, we first perform another comparison of the RA sizes between the now delayed partially observable repair case and the fully observable repair case in Fig. 8.20. As before, on the partially observable side, we use the best performing observability level 2.

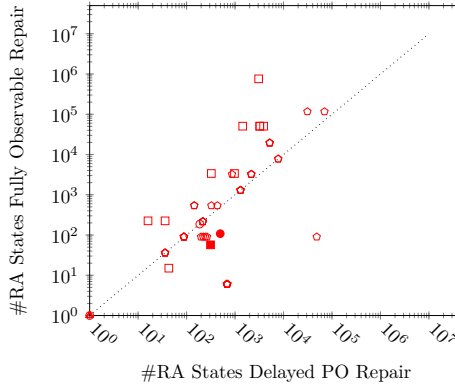


Fig. 8.20. Increase of RA for delayed partially observable repair vs fully observable repair case.

The number of instances where the fully observable case yields smaller RAs increases, especially for some instances from RC_{\circ} but otherwise replicates the pattern of the PO repair vs. fully observable repair case.

The detailed breakdown for each observability level is given in the plots in Fig. 8.21 on the following page. Since, on most observability levels, the number of solvable instances is small, there is not much to be evaluated. The notable cases are the two peaks observability levels 2 and 4. In Fig. 8.21c on the following page, we can see that on observability level 2, some of the RC_{\circ} instances scale well with the number of nodes, but all other families trend towards a TO. In Fig. 8.21e on the following page, we can see a similar result, except that now also RC_{\circ} quickly trends towards a TO.

8.4. PARTIALLY OBSERVABLE SCALABILITY EXPERIMENTS

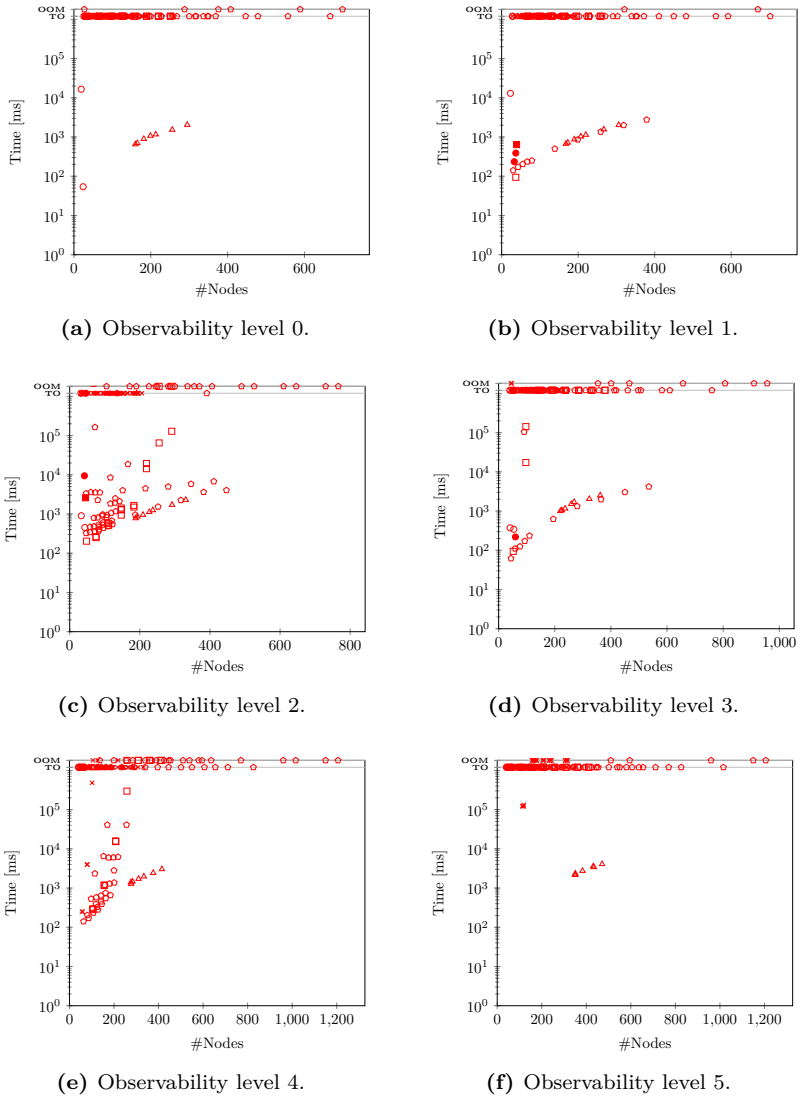


Fig. 8.21. Time measurement breakdown for observability levels 0-5.

8.5 Summary and Update of the NdDFT Hierarchy

In this section, we summarize the essential points of our findings and perform an update to the theoretical NdDFT hierarchy using our experimental results.

- **Modularization is key to performance.** Without modularization, the NdDFT approach does not scale. With modularization, the approach scales for a vast number of case studies.
- **Repair causes the RA sizes to explode.** We have seen a large number of OOMs caused by the RA composition and even seen partial observability having a positive performance impact due to reducing the RA sizes thanks to fewer observable events. Transferring orthogonal state space reduction onto the repair case might help mitigate the issue. Otherwise, it would be necessary to investigate a workflow that can drop the RA composition. However, the composition is at the latest required during the creation of the MC from the NdDFT and the RA to compute non-composable metrics such as the MTTF. The way forward might be not to use exact state spaces but approximation approaches to allow for composability.
- **Partial observability reduces scalability for non-repair cases but does not have much of an effect on repair cases.** While the concrete effects on the state space depend on the employed observability level, we have seen a strong negative impact of partial observability in the non-repairable case. Usually, this is driven by how effectively we can still perform modularization even under partial observability. A better, custom-tailored modularization approach might be a way forward to deal with this issue. On the other hand, for the repair case, we have not seen that many detrimental effects. Provided that at least enough levels remain observable to perform some basic modularization, adding partial observability to repair cases does not impact the RA synthesis negatively. Should an RA be synthesizable for a repairable NdDFT, it is likely that one can still synthesize an RA even after adding partial observability.
- **Delayed MONITOR gates should be avoided.** While not surprising, considering the complex state space fragment that gets involved from any partially observable event firing, the experiments have confirmed that the delayed MONITOR gate causes a state space increase that makes RA synthesis impossible. Even for the non-repairable case, on observability

level 5, we could only solve very few instances. Therefore, employing observation delays in MONITOR gates should only be done if the model absolutely requires them.

Since the delayed MONITOR gates lead to unscalable cases, we consider a refined version of the NdDFT class hierarchy from Fig. 6.1. We differentiate between the immediate PODFT class with no delayed MONITOR gates and the PODFT class as introduced in the theory section with delayed MONITOR gates. The updated hierarchy is shown in Fig. 8.22.

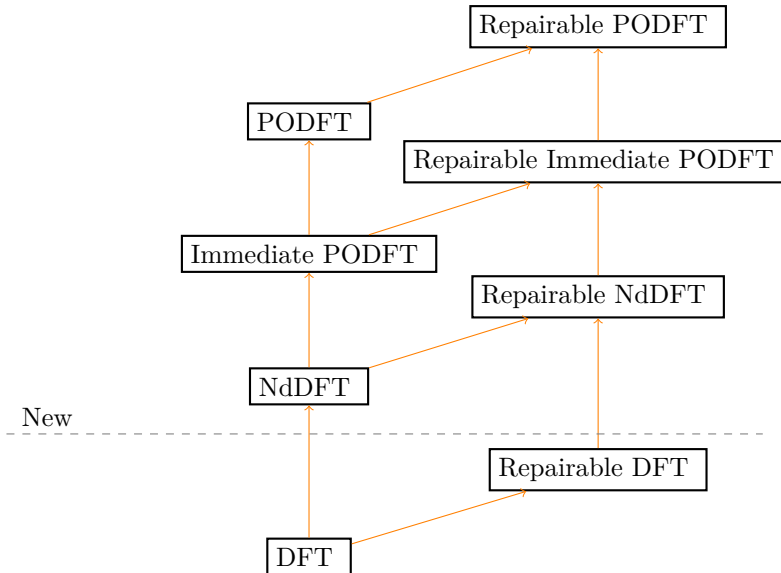


Fig. 8.22. Refined class hierarchy of DFT classes.

Chapter 9

Conclusion and Outlook

DFTs have rigid semantics, which leads to problems such as spare races. These rigid semantics make them difficult to apply to complex FDIR systems with features such as shared redundancies or partial observability.

In this thesis, we presented an FDIR model based on non-deterministic DFTs and deterministic recovery automata. Furthermore, we focused on the recovery automata synthesis problem to automatically generate an optimal recovery automaton with respect to a given metric from an NdDFT. The basic approach was to transform the NdDFT into a Markov Automaton, compute an optimal scheduler using model checking techniques, and extract the recovery automaton from the scheduler. The resulting algorithm was refined two-fold. Firstly, the state space of the synthesized automaton was reduced beyond what traditional trace-based equivalence is capable of. Secondly, modularization approaches were employed to avoid the state space explosion problem. We also introduced a partially observable DFT model (PODFTs) and lifted the previous constructions to this model.

We implemented the proposed approach within the open-source framework of Virtual Satellite and translated the formalized FDIR model into an FDIR conceptual data model. Furthermore, we proposed various implementation details for an efficient implementation.

Finally, we evaluated the scalability of our RA synthesis on case studies selected from the FFORT benchmark set. We were able to show that for the basic NdDFT model, our algorithm could solve nearly all literature case studies.

However, we also demonstrated that the efficiency was firmly bound to how well a model could be modularized. We could solve some instances for more complex classes, such as repairable NdDFTs and PODFTs. However, the results suggest that our approach would not scale up when modeling real-world FDIR systems for these advanced classes. Especially repair rates and delayed MONITOR gates were identified as problematic elements.

Overall, we set up a formal basis modeling complex FDIR systems using NdDFTs and PODFTs. Especially the benefit of including partial observability into the model seems promising, but it is not yet at a level where it can be applied for real-world spacecraft development. We discuss further ideas and propositions on how the approach could be further improved. Since all our Virtual Satellite implementation is open source, interested researchers can easily build upon the established research.

Outlook: Our work showed that modularization is a key to efficient RA synthesis. Therefore, improved techniques for dealing with instances where good modularization is not possible are of interest. Especially since an interconnected system with dependencies crossing between subsystems may be of particular interest for formal analysis. One possible angle of improvement would be to identify a weaker set of rules to identify a module, possibly with overlapping nodes. Since, at this step, we are not interested in the concrete metrical RAMS values but only in the correct RA behavior, it could be possible to identify rules that allow shared nodes to be included in multiple modules. Another direction would be to improve the efficiency of handling larger modules. In the past, symmetry reduction techniques have proved useful to combat the state space explosion problem in deterministic DFTs [62]. Leveraging these approaches to NdDFTs could be a promising approach to deal with larger modules.

Another central area of improvement could be to reduce the RA state space further. A generalization of the orthogonal merging rules to the repair case would be of interest. A prototypical generalization has been created in [13] and has also already been implemented. However, it is not clear how complete the proposed approach is. Rather than pursuing a generalization fixed on repairable NdDFTs, it might be more effective to pursue a generalization of orthogonal state space reduction where a given Linear Time Logic (LTL) formula describes the restrictions under which events are allowed to occur, similarly to fairness restrictions. Such a general approach could also allow to model further details, for example, that a cold spare can only fail if activated, and thus promises further avenues of reducing the state space of an RA.

Finally, on the area of partially observable NdDFTs, we reused our fully observable construction and encoded the beliefs in the second-stage belief Markov automaton construction. Comparing our two-stage construction approach to a representation using a partially observable Markov Automaton (POMA) would be interesting. Moreover, we applied a simple approach to lift modularization to the partially observable case by requiring the root node to be immediately observable. Since, as already stated, we have identified modularization as a critical factor for efficiency, investigating the possibility of a further refinement holds the potential to increase the scalability of PODFTs majorly.

Eidesstattliche Erklärung

Ich, Sascha Müller, erkläre hiermit, dass diese Dissertation und die darin dargelegten Inhalte die eigenen sind und selbstständig, als Ergebnis der eigenen originären Forschung, generiert wurden. Hiermit erkläre ich an Eides statt

1. Diese Arbeit wurde vollständig oder größtenteils in der Phase als Doktorand dieser Fakultät und Universität angefertigt;
2. Sofern irgendein Bestandteil dieser Dissertation zuvor für einen akademischen Abschluss oder eine andere Qualifikation an dieser oder einer anderen Institution verwendet wurde, wurde dies klar angezeigt;
3. Wenn immer andere eigene - oder Veröffentlichungen Dritter herangezogen wurden, wurden diese klar benannt;
4. Wenn aus anderen eigenen - oder Veröffentlichungen Dritter zitiert wurde, wurde stets die Quelle hierfür angegeben. Diese Dissertation ist vollständig meine eigene Arbeit, mit der Ausnahme solcher Zitate;
5. Alle wesentlichen Quellen von Unterstützung wurden benannt;
6. Wenn immer ein Teil dieser Dissertation auf der Zusammenarbeit mit anderen basiert, wurde von mir klar gekennzeichnet, was von anderen und was von mir selbst erarbeitet wurde;
7. Ein Teil oder Teile dieser Arbeit wurden zuvor veröffentlicht wie in [Sektion 1.2](#) aufgliedert.

Sascha Müller, 6. Juli 2023, Braunschweig

Erklärung zur Wahrung von Betriebsgeheimnissen

Ich gebe folgende Erklärung ab: Ich versichere, dass die in Zusammenarbeit mit dem Deutsche Zentrum für Luft- und Raumfahrt e. V. entstandene Dissertationsschrift durch ihre Veröffentlichung keine bestehenden Betriebsgeheimnisse verletzt.

Sascha Müller, 6. Juli 2023, Braunschweig

Bibliography

- [1] International Electrotechnical Commission, Geneva, Switzerland. *Fault Tree Analysis (FTA)*, 2006.
- [2] Enno Ruijters and Mariëlle Stoelinga. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer Science Review*, 15-16:29–62, 2015. doi:[10.1016/j.cosrev.2015.03.001](https://doi.org/10.1016/j.cosrev.2015.03.001).
- [3] Sascha Müller, Andreas Gerndt, and Thomas Noll. Synthesizing FDIR recovery strategies from non-deterministic dynamic fault trees. In *2017 AIAA SPACE Forum*, volume AIAA 2017-5163. American Institute of Aeronautics and Astronautics, 2017. doi:[10.2514/6.2017-5163](https://doi.org/10.2514/6.2017-5163).
- [4] Sascha Müller and Andreas Gerndt. Towards a conceptual data model for fault detection, isolation and recovery in Virtual Satellite. In *SECESA 2018*. European Space Agency, 2018. URL: <https://elib.dlr.de/122061/>.
- [5] Liana Mikaelyan, Sascha Müller, Andreas Gerndt, and Thomas Noll. Synthesizing and optimizing FDIR recovery strategies from fault trees. In *International Workshop on Formal Techniques for Safety-Critical Systems*, pages 37–54. Springer, 2018. doi:https://doi.org/10.1007/978-3-030-12988-0_3.
- [6] Sascha Müller, Andreas Gerndt, and Thomas Noll. Synthesizing failure detection, isolation, and recovery strategies from nondeterministic dynamic fault trees. *Journal of Aerospace Information Systems*, 16(2):52–60, 2019. doi:<https://doi.org/10.2514/1.I010669>.

- [7] Sascha Müller, Liana Mikaelyan, Andreas Gerndt, and Thomas Noll. Synthesizing and optimizing FDIR recovery strategies from fault trees. *Science of Computer Programming*, 196:102478, 2020. doi:<https://doi.org/10.1016/j.scico.2020.102478>.
- [8] Sascha Müller, Adeline Jordon, Andreas Gerndt, and Thomas Noll. A modular approach to non-deterministic dynamic fault trees. In *International Conference on Computer Safety, Reliability, and Security*, pages 243–257. Springer, 2021. doi:[10.1007/978-3-030-83903-1_16](https://doi.org/10.1007/978-3-030-83903-1_16).
- [9] Kilian Höflinger, Sascha Müller, Ting Peng, Moritz Ulmer, Daniel Lüdtke, and Andreas Gerndt. Dynamic fault tree analysis for a distributed onboard computer. In *2019 IEEE Aerospace Conference*, pages 1–13, 2019. doi:[10.1109/AERO.2019.8742128](https://doi.org/10.1109/AERO.2019.8742128).
- [10] Sascha Müller, Kilian Höflinger, Michal Smisek, and Andreas Gerndt. Towards an FDIR software fault tree library for onboard computers. In *2020 IEEE Aerospace Conference*, pages 1–10, 2020. doi:[10.1109/AERO47225.2020.9172756](https://doi.org/10.1109/AERO47225.2020.9172756).
- [11] Emanuel Kopp, Sascha Mueller, Fabian Greif, and Anko Boerner. Towards an H/W-S/W interface description for a comprehensive space systems simulation environment. In *2020 IEEE Aerospace Conference*, pages 1–14, 2020. doi:[10.1109/AERO47225.2020.9172440](https://doi.org/10.1109/AERO47225.2020.9172440).
- [12] Philipp M Fischer, Caroline Lange, Volker Maiwald, Sascha Müller, Andrii Kovalov, Janis Häseker, Thomas Gärtner, and Andreas Gerndt. Spacecraft interface management in concurrent engineering sessions. In *International Conference on Cooperative Design, Visualization and Engineering*, pages 54–63. Springer, 2019. doi:[10.1007/978-3-030-30949-7](https://doi.org/10.1007/978-3-030-30949-7).
- [13] Yogeswari Renganathan. Semantics of non-deterministic repairable fault trees. Master’s thesis, Technische Universität Darmstadt, 2019. URL: <https://elib.dlr.de/131219/>.
- [14] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004. doi:[10.1109/TDSC.2004.2](https://doi.org/10.1109/TDSC.2004.2).
- [15] A Wander and R Förstner. Innovative fault detection, isolation and recovery strategies on-board spacecraft: State of the art and research challenges.

- In *Deutscher Luft- und Raumfahrtkongress 2012*, Bonn, Germany, 2013. German Soc. for Aeronautics and Astronautics – Lilienthal-Oberth e.V. URL: <https://www.dglr.de/publikationen/2013/281268.pdf>.
- [16] Xavier Olive. FDI(R) for satellites: How to deal with high availability and robustness in the space domain? *International Journal of Applied Mathematics and Computer Science*, 22(1):99–107, 2012. doi:10.2478/v10006-012-0007-8.
- [17] Julien Marzat, Hélène Piet-Lahanier, Frédéric Damongeot, and Eric Walter. Model-based fault diagnosis for aerospace systems: a survey. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of aerospace engineering*, 226(10):1329–1360, 2012. doi:10.1177/0954410011421717.
- [18] O Benedettini, Tim S Baines, HW Lightfoot, and RM Greenough. State-of-the-art in integrated vehicle health management. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, 223(2):157–170, 2009. doi:10.1243/09544100JAERO446.
- [19] John Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, 34:43–98, 1956.
- [20] Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962. doi:10.1147/rd.62.0200.
- [21] Ali Zolghadri. Advanced model-based FDIR techniques for aerospace systems: Today challenges and opportunities. *Progress in Aerospace Sciences*, 53:18–29, 2012. doi:10.1016/j.paerosci.2012.02.004.
- [22] Domenico Reggio Patrick Bergner, André Posch. GAFE Methodology. 2018. URL: http://gafe.estec.esa.int/files/GAFE_Methodology.pdf.
- [23] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960. doi:10.1115/1.3662552.
- [24] Stephen Osder. Practical view of redundancy management application and theory. *Journal of Guidance, Control, and Dynamics*, 22(1):12–21, 1999. doi:10.2514/2.4363.

- [25] Walt Truskowski, Harold Hallock, Christopher Rouff, Jay Karlin, James Rash, Michael Hinchey, and Roy Sterritt. *Autonomous and autonomic systems: With applications to NASA intelligent spacecraft operations and exploration systems*. Springer Science & Business Media, 2009. doi:[10.1007/b105417](https://doi.org/10.1007/b105417).
- [26] Requirements & Standards Division, Noordwijk, Netherlands. *ECSS On-board control procedures*, 2012.
- [27] Requirements & Standards Division, Noordwijk, Netherlands. *ECSS Space segment operability*, 2008.
- [28] Jens Eickhoff. *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*. Springer Science & Business Media, 2011. doi:[10.1007/978-3-642-25170-2](https://doi.org/10.1007/978-3-642-25170-2).
- [29] Requirements & Standards Division, Noordwijk, Netherlands. *ECSS System - Glossary of terms*, 2012.
- [30] Richard E Barlow and Frank Proschan. *Mathematical theory of reliability*. SIAM, 1996. doi:[10.1126/science.148.3674.1208-a](https://doi.org/10.1126/science.148.3674.1208-a).
- [31] Marvin Rausand and Arnljot Høyland. *System reliability theory: models, statistical methods, and applications*, volume 396. John Wiley & Sons, 2004. doi:[10.1002/9780470316900.ch12](https://doi.org/10.1002/9780470316900.ch12).
- [32] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT press, 2008.
- [33] Christian Eisentraut, Holger Hermanns, and Lijun Zhang. On probabilistic automata in continuous time. In *IEEE Symposium on Logic in Computer Science*, pages 342–351. IEEE, 2010. doi:[10.1109/LICS.2010.41](https://doi.org/10.1109/LICS.2010.41).
- [34] Gianfranco Ciardo, Reinhard German, and Christoph Lindemann. A characterization of the stochastic process underlying a stochastic petri net. *IEEE Transactions on Software Engineering*, 20(7):506–515, 1994. doi:[10.1109/32.297939](https://doi.org/10.1109/32.297939).
- [35] Niklas Holsti and Matti Paakko. Towards advanced FDIR components. *Data Systems in Aerospace, DASIA 2001*, 2001.

- [36] Burton H Lee. Using Bayes belief networks in industrial FMEA modeling and analysis. In *Reliability and Maintainability Symposium, 2001. Proceedings. Annual*, pages 7–15. IEEE, 2001. doi:[10.1109/RAMS.2001.902434](https://doi.org/10.1109/RAMS.2001.902434).
- [37] Technical Committee and others, Geneva, Switzerland. *Analysis Techniques for System Reliability-Procedure for Failure Mode and Effects Analysis (FMEA)*, 2006.
- [38] William E Vesely, Francine F Goldberg, Norman H Roberts, and David F Haasl. Fault tree handbook. Technical report, Nuclear Regulatory Commission, Washington, DC, 1981. URL: <https://www.osti.gov/biblio/5762464-fault-tree-handbook>.
- [39] Olivier Coudert and Jean Christophe Madre. Fault tree analysis: 10²⁰ prime implicants and beyond. In *Reliability and Maintainability Symposium, 1993. Proceedings., Annual*, pages 240–245. IEEE, 1993. doi:[10.1109/RAMS.1993.296849](https://doi.org/10.1109/RAMS.1993.296849).
- [40] Ernest Edifor, Martin Walker, and Neil Gordon. Quantification of priority-or gates in temporal fault trees. In *International Conference on Computer Safety, Reliability, and Security*, volume 7612 of *LNCS*, pages 99–110. Springer, 2012. doi:[10.1007/978-3-642-33678-2_9](https://doi.org/10.1007/978-3-642-33678-2_9).
- [41] Sebastian Junges, Dennis Guck, Joost-Pieter Katoen, and Mariëlle Stoelinga. Uncovering dynamic fault trees. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, pages 299–310. IEEE, 2016. doi:[10.1109/DSN.2016.35](https://doi.org/10.1109/DSN.2016.35).
- [42] Joanne Bechta Dugan, Salvatore J Bavuso, and Mark A Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363–377, 1992. doi:[10.1109/24.159800](https://doi.org/10.1109/24.159800).
- [43] Mohammad Modarres, Mark P Kaminskiy, and Vasilii Krivtsov. *Reliability engineering and risk analysis: a practical guide*. CRC press, 2009.
- [44] Haiping Xu and Liudong Xing. Formal semantics and verification of dynamic reliability block diagrams for system reliability modeling. In *Proc. 11th International Conference on Software Engineering and Applications (SEA 2007)*, pages 155–162, 2007.
- [45] Michael Stamatelatos, Homayoon Dezfuli, George Apostolakis, Chester Everline, Sergio Guarro, Donovan Mathias, Ali Mosleh, Todd Paulos, David

- Riha, Curtis Smith, et al. Probabilistic risk assessment procedures guide for NASA managers and practitioners. 2011. URL: <https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20120001369.pdf>.
- [46] Peter Fenelon and John A McDermid. New directions in software safety: Causal modelling as an aid to integration. In *Workshop on Safety Case Construction, York (March 1994)*, 1992. URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.4673&rep=rep1&type=pdf>.
- [47] Hideo Nakano and Yoshiro Nakanishi. Graph representation and diagnosis for multiunit faults. *IEEE Transactions on Reliability*, 23(5):320–325, 1974. doi:10.1109/TR.1974.5215295.
- [48] SV Nageswara Rao and N Viswanadham. Fault diagnosis in dynamical systems: A graph theoretic approach. *International journal of systems science*, 18(4):687–695, 1987. doi:10.1080/00207728708964000.
- [49] Benjamin Bittner, Marco Bozzano, Alessandro Cimatti, Regis De Ferluc, Marco Gario, Andrea Guiotto, and Yuri Yushtein. An integrated process for FDIR design in aerospace. In *Model-Based Safety and Assessment*, volume 8822 of *LNCIS*, pages 82–95. Springer, 2014. doi:10.1007/978-3-319-12214-4_7.
- [50] Trevor A Kletz. *HAZOP and HAZAN: identifying and assessing process industry hazards*. CRC Press, 2001.
- [51] John A McDermid, Mark Nicholson, David J Pumfrey, and P Fenelon. Experience with the application of HAZOP to computer-based systems. In *Computer Assurance, 1995. COMPASS'95. Systems Integrity, Software Safety and Process Security. Proceedings of the Tenth Annual Conference on*, pages 37–48. IEEE, 1995. doi:10.1109/COMPASS.1995.521885.
- [52] Johan De Kleer and James Kurien. Fundamentals of model-based diagnosis. volume 36, pages 25–36. Elsevier, 2003. doi:10.1016/S1474-6670(17)36467-4.
- [53] Malcolm Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science*, 141(3):53–71, 2005. doi:10.1016/j.entcs.2005.02.051.
- [54] Xiaocheng Ge, Richard F Paige, and John A McDermid. Probabilistic failure propagation and transformation analysis. In *International Conference on*

- Computer Safety, Reliability, and Security*, volume 5775 of *LNCS*, pages 215–228. Springer, 2009. doi:[10.1007/978-3-642-04468-7_18](https://doi.org/10.1007/978-3-642-04468-7_18).
- [55] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. The COMPASS approach: Correctness, modelling and performability of aerospace systems. In *International Conference on Computer Safety, Reliability, and Security*, volume 5775 of *LNCS*, pages 173–186. Springer, 2009. doi:[10.1007/978-3-642-04468-7_15](https://doi.org/10.1007/978-3-642-04468-7_15).
- [56] Marco Beccuti, Giuliana Franceschinis, Daniele Codetta-Raiteri, and Serge Haddad. Computing optimal repair strategies by means of NdrFT modeling and analysis. *The Computer Journal*, 57(12):1870–1892, 2014. doi:[10.1093/comjnl/bxt134](https://doi.org/10.1093/comjnl/bxt134).
- [57] Enno Ruijters, Dennis Guck, Peter Drolenga, and Mariëlle Stoelinga. Fault maintenance trees: reliability centered maintenance via statistical model checking. In *Reliability and Maintainability Symposium (RAMS), 2016 Annual*, pages 1–6. IEEE, 2016. doi:[10.1109/RAMS.2016.7447986](https://doi.org/10.1109/RAMS.2016.7447986).
- [58] Daniele Codetta-Raiteri and Luigi Portinale. Dynamic Bayesian networks for fault detection, identification, and recovery in autonomous spacecraft. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 45(1):13–24, 2015. doi:[10.1109/TSMC.2014.2323212](https://doi.org/10.1109/TSMC.2014.2323212).
- [59] Daniele Codetta Raiteri and Luigi Portinale. ARPHA: an FDIR architecture for autonomous spacecrafts based on dynamic probabilistic graphical models. Technical Report TR-INF-2010-12-04-UNIPMN, Computer Science Institute, Università del Piemonte Orientale, Vercelli, Italy, 2010. URL: <http://www.di.unipmn.it/TechnicalReports/TR-INF-2010-12-04-UNIPMN.pdf>.
- [60] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. Safety, dependability and performance analysis of extended AADL models. *The Computer Journal*, 54(5):754–775, 2011. doi:[10.1093/comjnl/bxq024](https://doi.org/10.1093/comjnl/bxq024).
- [61] Sweewarman Balachandran and Ella Atkins. Markov decision process framework for flight safety assessment and management. *Journal of Guidance, Control, and Dynamics*, 40(4):817–830, 2017. doi:[10.2514/1.G001743](https://doi.org/10.2514/1.G001743).
- [62] Matthias Volk, Sebastian Junges, and Joost-Pieter Katoen. Advancing dynamic fault tree analysis-get succinct state spaces fast and synthesise failure rates. In *International Conference on Computer Safety, Reliability,*

-
- and Security*, volume 9922 of *LNCS*, pages 253–265. Springer, 2016. doi:
[10.1007/978-3-319-45477-1_20](https://doi.org/10.1007/978-3-319-45477-1_20).
- [63] Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, 4 edition, 2020.
- [64] Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and infinite-horizon partially observable markov decision problems. In *AAAI/IAAI*, pages 541–548, 1999. doi:
[10.1016/S0004-3702\(02\)00378-8](https://doi.org/10.1016/S0004-3702(02)00378-8).
- [65] Catherine Venturini, Barbara Braun, David Hinkley, and Greg Berg. Improving mission success of cubesats. In *Proceedings of the AIAA/USU Conference on Small Satellites*, 2018. URL: <http://digitalcommons.usu.edu/smallsat/2018/all2018/273/>.
- [66] G.H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955. doi:
[10.1002/j.1538-7305.1955.tb03788.x](https://doi.org/10.1002/j.1538-7305.1955.tb03788.x).
- [67] John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971. doi:
[10.1016/B978-0-12-417750-5.50022-1](https://doi.org/10.1016/B978-0-12-417750-5.50022-1).
- [68] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973. doi:
[10.1145/512927.512945](https://doi.org/10.1145/512927.512945).
- [69] Yves Dutuit and Antoine Rauzy. A linear-time algorithm to find modules of fault trees. *IEEE Transactions on Reliability*, 45(3):422–425, 1996. doi:
[10.1109/24.537011](https://doi.org/10.1109/24.537011).
- [70] Alexandru Mereacre, Joost-Pieter Katoen, Tingting Han, and Taolue Chen. Model checking of continuous-time markov chains against timed automata specifications. *Logical Methods in Computer Science*, 7, 2011. doi:
[10.2168/LMCS-7\(1:12\)2011](https://doi.org/10.2168/LMCS-7(1:12)2011).
- [71] Caroline Lange, Jan Thimo Grundmann, Michael Kretzenbacher, and Philipp Martin Fischer. Systematic reuse and platforming: Application examples for enhancing reuse with model-based systems engineering methods in space systems development. *Concurrent Engineering*, 26(1):77–92, 2018. doi:
[10.1177/1063293X17736358](https://doi.org/10.1177/1063293X17736358).

- [72] Sascha Müller. virtualsatellite/VirtualSatellite4-FDIR: Release 4.12.1, October 2020. doi:[10.5281/zenodo.6962365](https://doi.org/10.5281/zenodo.6962365).
- [73] Requirements & Standards Division, Noordwijk, Netherlands. *Space system data repository*, 2011. URL: <https://ecss.nl/hbstms/ecss-e-tm-10-23a-space-system-data-repository/>.
- [74] Philipp Martin Fischer, Meenakshi Deshmukh, Volker Maiwald, Dominik Quantius, Antonio Martelo Gomez, and Andreas Gerndt. Conceptual data model: A foundation for successful concurrent engineering. *Concurrent Engineering*, 26(1):55–76, 2018. doi:<https://doi.org/10.1177/1063293X17734592>.
- [75] Philipp M Fischer, Daniel Lüdtke, Caroline Lange, F-C Roshani, Frank Dannemann, and Andreas Gerndt. Implementing model-based system engineering for the whole lifecycle of a spacecraft. *CEAS Space journal*, 9(3):351–365, 2017. doi:<https://doi.org/10.1007/s12567-017-0166-4>.
- [76] Joanne Bechta Dugan, Kevin J Sullivan, and David Coppit. Developing a low-cost high-quality software tool for dynamic fault-tree analysis. *IEEE Transactions on Reliability*, 49(1):49–59, 2000. doi:[10.1109/24.855536](https://doi.org/10.1109/24.855536).
- [77] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.
- [78] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. In *International Conference on Computer Aided Verification*, pages 592–600. Springer, 2017. doi:[10.1007/978-3-319-63390-9_31](https://doi.org/10.1007/978-3-319-63390-9_31).
- [79] Dennis Guck, Jip Spel, and Mariëlle Stoelinga. Dftcalc: Reliability centered maintenance via fault tree analysis (tool paper). In *International Conference on Formal Engineering Methods*, pages 304–311. Springer, 2015. doi:[10.1007/978-3-319-25423-4_19](https://doi.org/10.1007/978-3-319-25423-4_19).
- [80] Enno Ruijters, Carlos E Budde, Muhammad Chenariyan Nakhaee, Mariëlle Ida Antoinette Stoelinga, Doina Bucur, Djoerd Hiemstra, and Stefano Schivo. FFORT: A benchmark suite for fault tree analysis. pages 878–885, 2019. doi:https://doi.org/10.3850/978-981-11-2724-3_0641-cd.