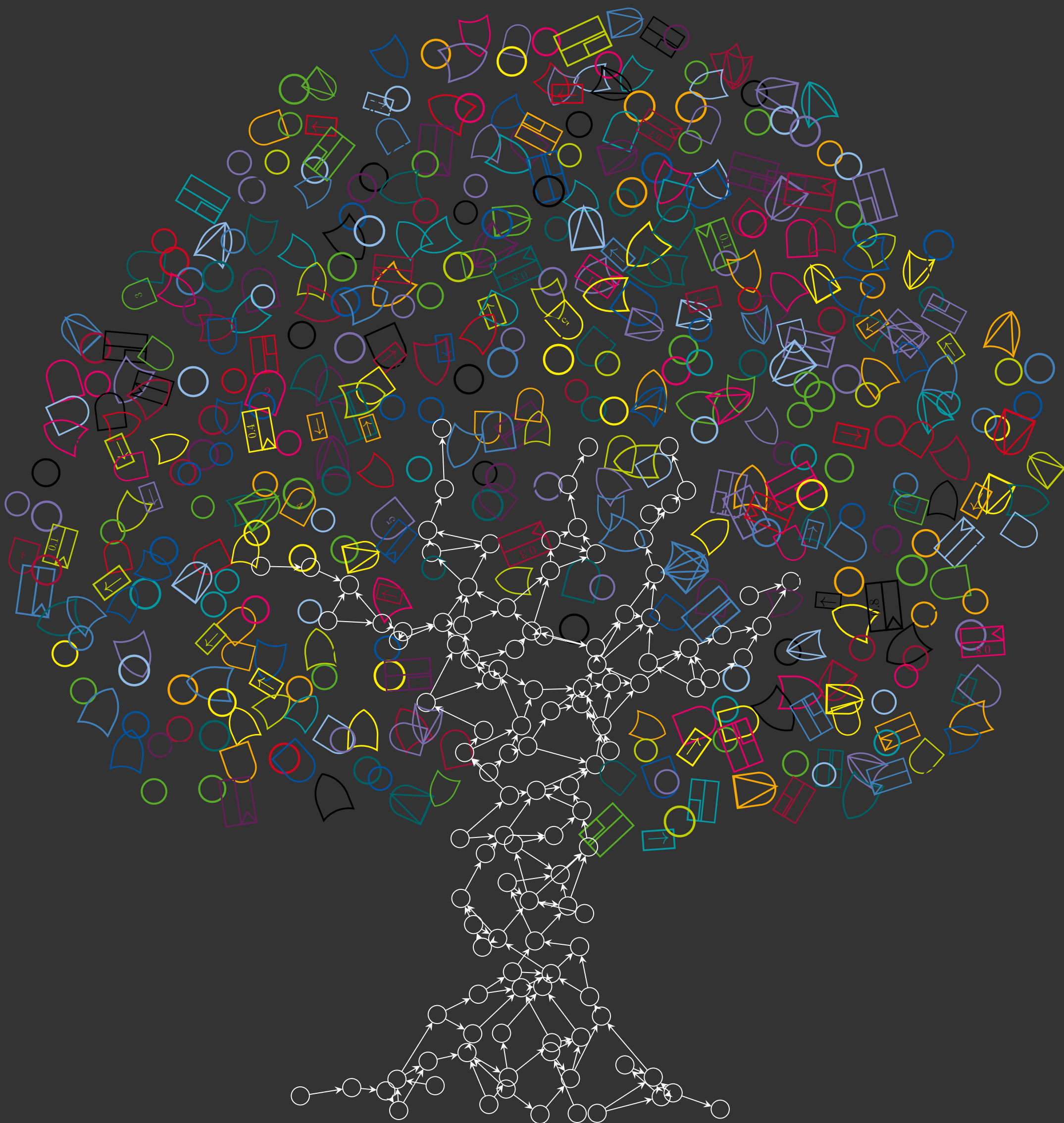
Dynamic Fault Trees

Semantics, Analysis and Applications



Matthias Volk

Dynamic Fault Trees: Semantics, Analysis and Applications

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

Matthias Volk, M. Sc.
aus Greven

Berichter: Prof. Dr. Ir. Dr. h.c. Joost-Pieter Katoen
P.D. Alessandro Cimatti, MSc.

Tag der mündlichen Prüfung: 28. April 2022

Abstract

Safe and reliable systems are crucial in today's society. Failures can have severe consequences, ranging from performance degradation over environmental and economic costs to loss of human lives. The growing number of systems, the rise of cyber-physical systems and the connectedness of components lead to an ever increasing complexity.

A broad range of approaches and formal models have been proposed to analyse and improve the reliability of systems. *Fault trees* are a prominent and widely-used model in reliability engineering. They model how component failures propagate through a system and lead to a failure of the overall system. While fault trees are widely adopted in industry, their static nature discards dynamic behaviour present in modern systems. *Dynamic fault trees* (DFTs) are an extension of static fault trees and allow more modelling flexibility by introducing dynamic gates, spare management, functional dependencies and failure restrictions.

In this thesis, we investigate dynamic fault trees in detail and consider three main aspects: (1) the precise semantics of DFTs, (2) the analysis of DFTs by model checking techniques, and (3) the application of DFTs in the automotive and railway domain.

In the first part we specify the *semantics* of dynamic fault trees. We present a precise definition of the semantics for each DFT element in terms of *generalized stochastic Petri nets (GSPNs)*. We also investigate multiple semantic questions resulting from the combination of DFT elements. Our resulting framework based on GSPNs subsumes the major existing DFT semantics and allows to pinpoint their differences.

The second part presents *analysis techniques* for DFTs. Our analysis is based on probabilistic model checking and uses *Markov automata* as the underlying model. The analysis exploits efficient, off-the-shelf algorithms for model checking and allows to compute a broad range of measures which go far beyond typical reliability. We present several (orthogonal) optimisation techniques which exploit symmetries, irrelevant failures and independent subtrees to improve the state-space generation times. Additionally, we develop an approximation algorithm based on partial state-space exploration which iteratively refines the state space until the desired precision of the result is reached. All presented approaches are implemented in the open-source model checker Storm and are evaluated on a DFT benchmark suite. The evaluation shows that our tool Storm-dft is state-of-the-art for DFT analysis.

The third part presents the *application* of DFTs in both (1) the automotive and (2) railway domain. The first case study considers a vehicle guidance system for autonomous driving. We model several safety concepts and analyse different partitionings of functions on hardware. The resulting models are among the largest DFTs to date. The second case study considers train routing options in railway station areas in terms of available infrastructure elements. The DFTs are automatically generated from the given infrastructure data and train path information. We analyse how switch failures impact the potential train routes in a station and determine the most critical components.

Zusammenfassung

Sichere und zuverlässige Systeme sind von entscheidender Bedeutung in der heutigen Zeit. Ausfälle können schwerwiegende Folgen haben und reichen von Leistungseinbußen über ökologische und wirtschaftliche Schäden bis hin zum Verlust von Menschenleben. Die wachsende Anzahl von Systemen, die Zunahme von cyber-physischen Systemen und die breite Vernetzung von Komponenten führen zu einer immer größer werdenden Komplexität.

Eine Vielzahl von Ansätzen und formalen Modellen ist vorgeschlagen worden, um die Zuverlässigkeit von Systemen zu analysieren und zu verbessern. *Fehlerbäume*[1] sind ein bekanntes und weit verbreitetes Modell in der Zuverlässigkeitsanalyse. Fehlerbäume modellieren, wie sich Ausfälle von Komponenten in einem System ausbreiten und zu einem Ausfall des Gesamtsystems führen. Während Fehlerbäume in der Industrie weit verbreitet sind, berücksichtigt ihre statische Natur nicht das dynamische Verhalten, welches in modernen Systemen vorhanden ist. *Dynamische Fehlerbäume (DFTs)* sind eine Erweiterung von statischen Fehlerbäumen und ermöglichen mehr Flexibilität in der Modellierung durch die Einführung von dynamischen Gattern, Management von Reserveteilen, funktionalen Abhängigkeiten und Einschränkungen von Ausfällen.

In dieser Arbeit untersuchen wir dynamische Fehlerbäume im Detail und betrachten drei Hauptaspekte: (1) die präzise Semantik von DFTs, (2) die Analyse von DFTs mit Hilfe von Model-Checking-Techniken und (3) den Einsatz von DFTs im Automobil- und Eisenbahnbereich.

Im ersten Teil spezifizieren wir die *Semantik* von dynamischen Fehlerbäumen. Wir präsentieren eine präzise Definition der Semantik für jedes DFT-Element in Form von *generalisierten stochastischen Petri-Netzen (GSPNs)*. Außerdem untersuchen wir mehrere semantische Fragestellungen, die sich aus der Kombination von verschiedenen DFT-Elementen ergeben. Unser Modell auf Basis von GSPNs umfasst die wichtigsten existierenden DFT-Semantiken und erlaubt es, ihre Unterschiede genau zu bestimmen.

Der zweite Teil stellt *Analysetechniken* für DFTs vor. Unsere Analyse basiert auf Model-Checking für probabilistische Systeme und verwendet *Markov-Automaten* als Basis-Modell. Die Analyse nutzt effiziente Standard-Algorithmen für das Model-Checking und ermöglicht es, eine breite Auswahl von Metriken zu berechnen, die weit über die typischerweise berechnete Zuverlässigkeit hinausgehen. Wir stellen mehrere (orthogonale) Optimierungen vor, die unter anderem Symmetrien, irrelevante Ausfälle und unabhängige Teilbäume ausnutzen, um die Generierung des Zustandsraums zu beschleunigen. Zusätzlich entwickeln wir einen Approximationsalgorithmus, der nur Teile des Zustandsraums generiert. Der Zustandsraum wird dabei solange iterativ verfeinert, bis die gewünschte Genauigkeit des Ergebnisses erreicht ist. Alle vorgestellten Ansätze werden in dem Open-Source-Tool Storm implementiert und auf

[1] Manchmal auch Fehlzustandsbäume genannt.

einer Sammlung von DFT-Beispielen evaluiert. Die Evaluierung zeigt, dass unser Tool Storm-dft der aktuelle Stand der Technik für die DFT-Analyse ist.

Im dritten Teil präsentieren wir den *Einsatz* von DFTs im (1) Automobil- und (2) Eisenbahnbereich. Die erste Fallstudie betrachtet ein Fahrzeugführungssystem für autonomes Fahren. Wir modellieren verschiedene Sicherheitskonzepte und analysieren verschiedene Zuordnungen von Funktionen auf die Hardware. Die resultierenden Modelle gehören zu den bisher größten analysierten DFTs. Die zweite Fallstudie betrachtet die Zugführung in Bahnhofsbereichen unter Berücksichtigung der verfügbaren Infrastrukturelemente. Die DFTs werden automatisch aus den gegebenen Infrastrukturdaten und Trasseninformationen generiert. Wir analysieren, wie sich Weichenausfälle auf die möglichen Zugrouten in einem Bahnhof auswirken und ermitteln die kritischsten Komponenten.

Acknowledgements

When I started my PhD journey, my research goal was to develop and improve approaches for parameter synthesis in probabilistic models. Dynamic fault trees were supposed to be one of the application areas for this purpose. Working together with Joost-Pieter and Sebastian, we saw however that we first needed to improve the translation from DFTs to Markov models before even considering any analysis. We therefore set out to develop and implement techniques for faster DFT analysis. Along the way, we encountered more possibilities for improvements and many more open questions. In the end, my research focus diverged from the original plan and I fully focused on DFTs in all their facets. But I think that is the beauty of science: discovering new questions along the way and being able to follow these ideas to—hopefully interesting—results and insights.

I am very grateful to Joost-Pieter for his great supervision during my PhD. You gave me lots of freedom to find my own way and explore the research area while always providing much appreciated guidance. I learned so much during our discussions which not only helped me in my research and academic life but also for my personal development. I am thankful for your support, encouragement and patience, in particular when I was making mistakes or exceeded deadlines—in particular regarding the mean-time-to-finish[2]. I also appreciate that you always took the opportunity to advertise our work around the world and it was encouraging to see that our work could make an impact. Lastly, I want to thank you for fostering a great atmosphere and research environment at the chair.

I am thankful to Alessandro Cimatti for spending much of his valuable time reading and reviewing my thesis, and for interesting discussions and suggestions regarding DFTs. I also want to thank Wil van der Aalst and Stefan Kowalewski for being part of my thesis committee.

Research is very much a team effort and I would never have gotten anywhere without the endless input from and discussions with my colleagues and co-authors. I am thankful to my co-authors Erika Ábrahám, Saba Aflaki, Borzoo Bonakdarpour, Marc Bouissou, Alexis Braibant, Yassmeen Elderhalli, Majdi Ghadhab, Osman Hasan, Christian Hensel, Nils Jansen, Sebastian Junges, Joost-Pieter Katoen, Shahid Khan, Matthias Kuntz, Nils Nießen, Tim Quatmann, Falak Sher, Mariëlle Stoelinga, Arne Storjohann, Sofiène Tahar, Norman Weik and Muhammad Ahmad Zafar. Together, we developed ideas and transformed them into tangible results. I enjoyed our collaborations, had fun working together and learned a lot from you.

Working at the chair was always a pleasure for me, because my colleagues made it a very enjoyable time. You helped me with research and teaching, offered valuable

[2] https://www.gocomics.com/calvinandhobbes/1992/05/21

suggestions and feedback, and made a lasting impact in my life. I enjoyed our quick or longer chats over a coffee, lunch time discussions over basically any possible topic, passionate duels in table football, after-work beers and of course our retreats to Söllerhaus. Alas, five years is far too short a time to work among such excellent and admirable colleagues. I want to thank Joost-Pieter Katoen, Thomas Noll, Kevin Batz, Jana Berger, Harold Bruintjes, Souymodip Chakraborty, Mingshuai Chen, Ira Fesefeldt, Christian Hensel, Christina Jansen, Nils Jansen, Sebastian Junges, Mojgan Kamali, Benjamin Kaminski, Shahid Khan, Lutz Klinkenberg, Tim Felix Lange, Anna Lukina, Christoph Matheja, Joshua Moerman, Federico Olmedo, Tim Quatmann, Bahare Salmani, Jip Spel, Marcin Szymczak, Tobias Winkler and Hao Wu from the MOVES group; Erika Ábrahám, Xin Chen, Florian Corzilius, Rebecca Haehn, Gereon Kremer, Francesco Leofante, Ulrich Loup, Johanna Nellen, Pascal Richter and Stefan Schupp from the THS group; Jürgen Giesl, Daniel Cloerkes, Stefan Dollase, Florian Frohn, Marcel Hark, Jera Hensel and David Keller from LuFG i2; and all the members of the UnRAVeL RTG. A special thanks to Helen Bolke-Hermanns, Elke Ohlenforst and Birgit Willms for all your assistance regarding organisational matters, offering helpful tips navigating bureaucracy and doing so many task in the background we take for granted.

I want to give a special thanks to my fellow 'Stormtroopers'.

Chris, I am glad that you started all the work on Storm which we now profit from. You encouraged me early on to implement things in Storm on my own and were always quick to guide me in the right direction. While we might disagree on the Mensa-vs-bakery question, we were always a good team when it came to table football.

Sebastian, you introduced me to fault trees in the first place and your Master thesis was a great source of knowledge. I enjoyed very much working and researching together with you. You have seemingly endless ideas and were a constant source of inspiration. I spent countlessly many great times with you: working with Lego Mindstorms, late-night work before submission deadlines, Mama Tai dinners, developing and maintaining all the software, supervising students and lots of research discussions.

Tim, I entered your office many, many times and disturbed you in your work by asking Storm-related or other questions. But I could always count on you to provide helpful comments and solutions, or even implement them for me. I enjoyed our conversations and being able to always have a chat with you—and then squeezing past the door and plant back to my office.

Jip, you knew from the beginning that speaking Dutch is a valuable skill. You taught me a lot and we could talk hours about shared interests—sports, languages, sports, Lego and sports. You were always quick to encourage and help me, and I appreciate our personal conversations.

Thanks to Christina for sharing an office with me. You helped me a lot when I started at the chair and introduced to me to all important things—including Ganymed.

Shahid, you were a great office mate and I admire the drive you have. I was happy to share my knowledge with you and see where things were still missing or could be improved. At the same time learned a lot of new things from you as well.

I am very thankful for Erika, Nils and Florian in introducing me to research and their research group early on. You showed me how interesting and how much fun research is. I appreciate the opportunities you gave me very early in my studies.

I had a great time as member of the UnRAVeL RTG. In particular, I want to thank Nils Nießen, Norman Weik and Stephan Zieger for introducing me to railway systems and teaching me so many things about railway safety. Norman, I enjoyed our collaboration, interesting discussions and making videos about our research.

I was lucky enough to work with great and smart student assistants over the years. Thank you to Daniel Basgöze, Alexander Bork, Dustin Jungen, Lutz Klinkenberg and Hannah Mertens. You helped me a lot by envisioning, implementing, extending, improving and evaluating approaches, and I am grateful for all your efforts and our interesting discussions. I hope you could experience for yourself how much fun and how interesting research can be.

I am also happy to have (co-)supervised Bachelor and Master thesis projects. Thanks to Alexander, Christopher, Daniel, Dustin, Fynn, Hannah, Markus, Michael and Ronja for performing good research, progressing the status-quo and showing me that new directions of research were promising.

I also want to thank Mariëlle for giving me the opportunity to continue my research at the UT. In particular, I am grateful for your patience when finishing the thesis took longer than planned.

All this work is of course based on the giants before me. I am thankful for all the research performed by much brighter people which I could use for my own research. I am also very thankful for the countless developers and maintainers of open-source software I use nearly every minute of my life. Without their—often unpaid—work, my work would have been impossible.

Outside of work, I am grateful for my good friends Thomas, Katharina, Jan, Christian and Stefan. You reminded me that work is not everything and I enjoyed spending time with you. Thank you for your constant support, the many deep conversations we had throughout the years and for creating many wonderful memories together.

Ich bin meinen Eltern und Brüdern unendlich dankbar. Barbara, Thomas, Stephan und Christian, ihr seid immer für mich da, egal in welcher Lebenssituation. Ihr unterstützt mich bedingungslos, steht mir mit Ratschlägen zur Seite und bietet mir immer einen willkommenen Wohlfühlort. Ich bin sehr glücklich, so eine tolle Familie zu haben.

Contents

Part I

Introduction

1 Introduction

Reliable systems operating without failure are essential in our society. Failures of systems can have severe consequences, ranging from performance degradation over environmental and economic costs to loss of human lives.

Physical systems can fail due to a multitude of reasons such as wear over time, incorrect operation, inhospitable working environments, cascading effects of smaller faults, and many more. For instance, the space shuttle Challenger disaster in 1986 was caused by faulty O-ring seals which could not handle the cold conditions present during the launch [Pre86]. The space shuttle Columbia disaster in 2003 was caused by damage to the heat shield—resulting from a piece of foam insulating material [Col03].

In 2011, the nuclear accident at the Fukushima nuclear power plant happened even though the initial safety measures worked as intended [Inv12]. After an earthquake, the electricity supplies for the nuclear reactors failed, but the backup diesel generators took over as intended. However, the tsunami caused by the earthquake was higher than anticipated in the designs. The wave swept over the sea barriers and flooded the diesel generators located in the basement. As a result, the reactor cooling failed and lead to a meltdown.

The two fatal crashes of planes from type Boeing 737 MAX in 2018 and 2019 were caused by malfunctioning of the Maneuvering Characteristics Augmentation System (MCAS) [Fed20a]. In particular, the design contained a single point of failure: 'Erroneous data from a single AOA [angle-of-attack] sensor activated MCAS and subsequently caused airplane nose-downtrim of the horizontal stabilizer.' [Fed20a]

Apart from catastrophes which make headlines, most failures in daily operations are non-lethal due to the existing rigorous safety mechanism in place. Nevertheless, failures can still lead to degraded operations or loss of functionality. For example, in the German railway system in 2019, 13 % of all delays in long-distance trains were caused by failures of the trains themselves and 7 % were caused by failures in the signalling and interlocking [Bun20].

Ensuring reliability in a system is of utmost importance. This can be achieved either through prevention of faults—if possible—or mitigation once a fault has occurred—for example by designing fail-safe systems or introducing redundancies. The growing number of systems, the rise of cyber-physical systems and the connectedness of components have lead to an ever increasing complexity and thus, also made the task of designing reliable systems more complex.

1.1 Fault Trees

An important step in creating reliable systems is the analysis of the system's reliability in the first place. One prominent approach for reliability analysis is *fault tree analysis (FTA)* [SVDF+02; RS15; TB17].

fault tree analysis (FTA)

1.1.1 Static Fault Trees

Fault trees (FTs) were first introduced in 1961 [Wat61] by H. Watson from Bell Labs along with A. Mearns to analyse the Minuteman missile system [Eri99]. Shortly after, Boeing adopted fault trees for their civil aircrafts [Hix68]. To date, fault tree analysis has been widely adopted in industry and is specified for example in an IEC standard [IEC61025]. Fault tree analysis is required in several safety-critical domains: for aircrafts by the Federal Aviation Administration (FAA) [Fed20b], for nuclear power plants by the Nuclear Regulatory Commission (NRC) [VGRH81] and in the automotive domain by the ISO standard 26262 [ISO26262].

Fault trees model how failures of (sub-)components propagate through the system and eventually lead to a system failure. The creation of a fault tree follows a top-down approach. It starts with the *top-level element* which represents the failure of the complete system. The system failure is subdivided into failures of sub-components—which can then be further subdivided. This hierarchical approach is continued up to the desired level of detail. The leaves in the resulting tree represent components which are not further subdivided, so called *basic events (BEs)*. A basic event fails according to its associated probability distribution.

If a BE fails, the failure is propagated upwards through the fault tree. The intermediate nodes–so called *gates*—fail themselves if their failure condition is satisfied. For instance, gates of type AND fail if all their sub-components have failed whereas gates of type OR fail if at least one of their sub-components has failed. Fault trees thus represent Boolean functions over the BEs encoding the system's failure conditions.

Example 1.1 (Fault tree for laptop) Consider the FT in Figure 1.1 modelling the (simplified) failure behaviour of a laptop. The failure of the laptop is subdivided into the failures of the operating system (OS), the processors and the power. The OR-gate models that if either one of the three components fails, the laptop fails. Each component can be further subdivided. For example, the OS fails if both the installed OS and the backup—from which the OS could be recovered—fail. This is modelled with an AND-gate. The backup itself fails, if an internal error in the backup occurs or if the hard disk fails on which the backup is stored. Both types of error are basic events and therefore not further subdivided. In our example, the backup is stored on the same hard disk as the installed OS. As a result, a failure of

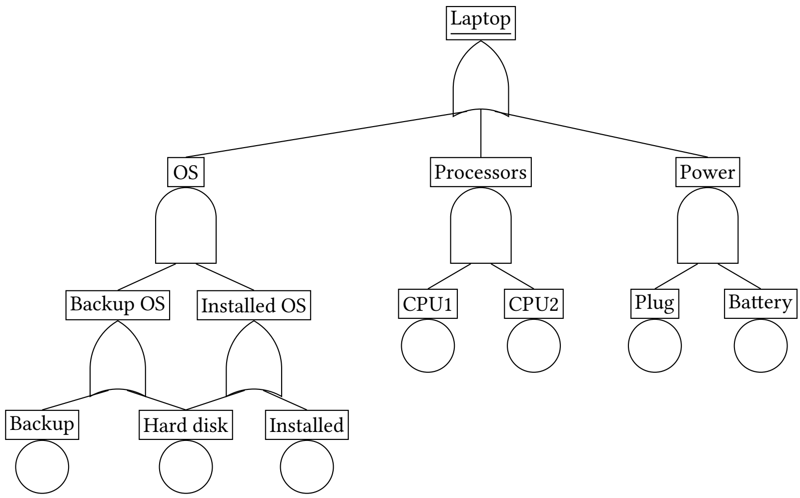


Figure 1.1: (Static) fault tree modelling a simplified laptop

> the hard disk will render both the installed OS and the backup failed.

Analysis techniques | Analysis of fault trees is usually performed by examining the so-called *cut sets*: sets of BEs where the failure of all components in a cut set leads to system failure. Cut sets are most often computed by representing a fault tree as a *binary decision diagram (BDD)* [Rau93; CM93]. For a detailed overview of different analysis techniques for fault trees, see [RS15].

1.1.2 Dynamic Fault Trees

The order in which components fail is irrelevant for the system failure in a fault tree. Fault trees with AND- and OR-gates are thus often referred to as *static fault trees (SFTs)*. While SFTs are widely used in practice, they are not equipped to faithfully model more complex—dynamic—behaviour present in modern systems. For example, the following three behaviours cannot be adequately modelled with standard (static) fault trees.

Ordered failures | The order in which components fail can be important. If for example the main component in a redundant system fails, the operation is switched from this component to a backup component. If both the main component and the switch are failed, the system fails as the backup component cannot be activated through the switch. However, if first the main component fails while the switch is still working, the backup component can be activated as intended. A failure of the switch later on—after already performing the designed task—therefore has no effect. In summary, we need the possibility to distinguish between a failure of the main component after the switch failure—where no switching can take place—and a failure of the main component before the switch failure—which has no immediate effect.

Spare management | Back-up components for redundant systems can have different failure probabilities depending on whether they are actively used or not. As soon as the main component of a redundant system fails, the back-up component is used and thus, might have an increased failure probability. Additionally, such back-up—or spare—components might replace one of several components. Think for example of the spare tire in a car which can replace any of the four tires. Using the spare tire as replacement for one of the tires also renders it unavailable for the other three.

Functional dependencies | Complex systems often contain functional dependencies between different components. One example is a power outage which affects several components which might otherwise not be in relation. Additionally, cyclic dependencies where several components depend on each other need to be modelled as well. Cycles in particular are hard to model in the tree-structure of static fault trees.

Dynamic fault trees | To overcome these limitations, several extensions of static fault trees have been proposed in the past. One of the most prominent extension is *dynamic fault trees (DFTs)*. Dynamic fault trees were first proposed by Dugan, Bavuso and Boyd [DBB90] in 1990 and subsequently developed [Boy91; DBB92]. DFTs extend classical SFTs by introducing several new types of gates:

- priority-AND (PAND) and sequence enforcer (SEQ) for order-dependent failures,
- SPARE-gate for spare management and activation, and
- functional dependency (FDEP).

The additional types of gates in DFTs increase the expressiveness and allow for more faithful modelling compared to SFTs. As a result, the analysis results computed on a DFT model more accurately reflect the reality.

We are using DFTs throughout this thesis as the main model for reliability analysis.

> **Example 1.2 (Dynamic fault tree for laptop)** We extend the (static) fault tree from Figure 1.1 and present a corresponding dynamic fault tree in Figure 1.2.
> While the general structure and the basic events remain the same, we use the dynamic gates of DFTs for a more faithful modelling. For example, the power component is modelled with a SPARE-gate now. As long as the power is provided by the plug, the battery is in a passive mode and has a reduced failure rate.
> The failure of the OS can also be modelled in more detail with a priority-AND

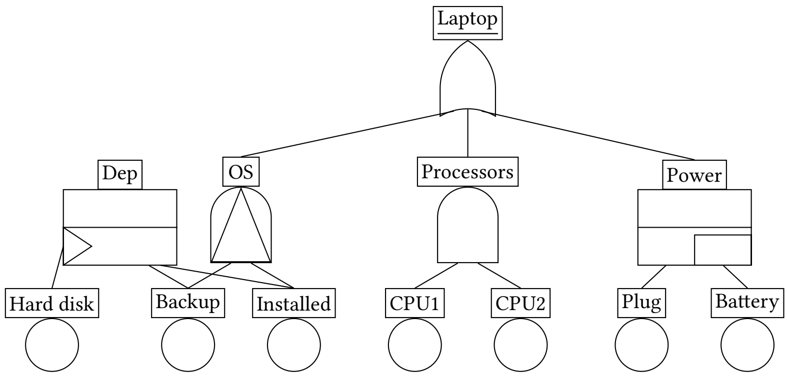


Figure 1.2: Dynamic fault tree modelling a simplified laptop

(PAND). A PAND fails if all its sub-components have failed in order from left to right. In our case, if the backup fails before the installed system, there is no possibility to recover the OS and the OS fails. However, if first the installed system fails, then the OS can still be recovered from the backup and the OS does not fail. Lastly, both the backup and the installed OS are functionally dependent on the hard disk. This is modelled with the functional dependency (FDEP). If the hard disk fails, then both the backup and the installed OS are also rendered failed.

Application areas | Dynamic fault trees have been adopted in industry and are used in a wide range of application areas including nuclear power plants [RGRK+09; GLYZ+15b], avionics [DBB92; Mog19], aerospace [SVDF+02; HMPU+19], automotive engineering [Sch09] and railway systems [RGNS16; RGDP+16].

Analysis techniques | Analysis of dynamic fault trees was originally performed by translating them into Markovian models [DBB92]. Joanne Bechta Dugan and others developed several improvements [GD97; MDCS98] and tools [DVG97] for this Markovian analysis, culminating in the Galileo tool [SDC99]. The compositional approach of [BCS10] also translates DFTs into Markov models and is implemented in the tool DFTCalc [ABBG+13]. Other DFT analysis methods are based on Bayesian networks [MPB05; BD05a] or Monte Carlo simulation [RGRK+09; RRBS19]. We refer to [RS15] and [KSYY20] for general overviews of existing DFT analysis techniques.

Remark 1.1 (No repairs) Note that in this thesis, we are focusing on failures only and are not considering any maintenance actions or *repairs*.

1.2 Motivation

In the context of safety-critical systems, analysing the systems and assessing their reliability is crucial for implementing the necessary safety precautions. Otherwise, failures of components can have dramatic consequences—see the historic examples from before. Maybe even more important is the guarantee that the obtained analysis results are correct and can be trusted. Invalid results lead to a false sense of security and might even prevent the implementation of additional safety measures.

> 'A serious risk created by our growing dependence on modeling tools is that engineers will make adverse decisions based on computed results that are not valid. If the cost of such a decision is potentially large and if there is no sound basis for assessing validity, then it is uninformed and even unethical for an engineer to accept such results.'
> (Coppit, Sullivan and Dugan [CSD00])

In our context of dynamic fault trees, invalid results can stem from three main sources:

1. unclear semantics leading to models with unintended behaviour,
2. incorrect or numerically imprecise computation of the analysis results, and
3. inadequate modelling where the DFT does not fully represent the real-life system.

In this thesis, we tackle all three problems and improve on the existing state-of-the-art.

1.2.1 Semantics

Problem | The semantics of static fault trees is clearly defined. The failure of the top-level event can for example be expressed as a Boolean function. In dynamic fault trees in contrast, the dynamic nature of the additional gates makes a clear definition of the semantics more complicated [CSD00]. While each dynamic gate on its own seems to have a concise definition, the interplay between different gates leads to intricate issues which are often overlooked and not properly defined in the literature [JGKS16].

One example is the PAND-gate which fails if its sub-elements fail in order from left to right. However, what is the behaviour if multiple sub-elements fail at the same time? Another issue is the order in which failures are propagated through the DFT. In certain cases, different failure propagation orders can lead to different results. That means whether a DFT element is failed or not depends on the order in which component failures are propagated through the DFT. We like to emphasize that this is not an artificial problem as existing tools for DFT analysis use slightly different DFT semantics. As a result, current tools will return conflicting analysis results for specific DFTs—even though all internal calculations within each tool are correct.

Contribution | Our aim is to capture and compare the existing DFT semantics. To this end, we introduce a thorough semantic of DFTs in terms of *generalized stochastic Petri nets (GSPNs)* [MBCD+95]. The behaviour of each DFT element is defined independently in terms of a GSPN. Composing the individual GSPNs yields a complete GSPN representing the behaviour of the overall DFT. We prove the correctness of our GSPN representation and show that all major existing DFT semantics can be captured by our GSPN framework. The differences between the semantics can be pinpointed to using different priorities and different handling of non-determinism in the GSPN. Additionally, we investigate the complex interplay of DFT gates and present several semantic intricacies present in DFTs. Our work helps in giving a precise and distinct meaning to DFT models and thus, helps practitioners be certain about the modelled behaviour of a system.

1.2.2 Analysis

Problem | The analysis of DFTs should return trustworthy results, ideally exact results without any errors. Common analysis approaches such as using Bayesian networks [BD05a; MPB05] or Monte Carlo simulation [RGRK+09; RRBS19] suffer from numerical imprecisions due to discretization or only provide results with certain confidence. As DFTs are used in the context of safety-critical systems, such imprecisions might not be acceptable. A counter measure is to use approaches guaranteeing precise results. The most prominent approach is *probabilistic model checking*.

Probabilistic model checking | Model checking [BK08; CHVB18] is an automated technique which checks whether a given property holds for a given model. The check is performed by systematically exploring all possible states in the system. Probabilistic model checking [Kwi03; Kat16] extends this approach to probabilistic models containing uncertainty. In our setting, the models are Markov chains [Nor98; Ste09], more precisely *Markov automata* [EHZ10]. Probabilistic model checking is a mature analysis technique [HHHK+19; BHKK+21] supported by dedicated tools such as Modest [HH14], PRISM [KNP11] or Storm [12].

Contribution | Following the lines of previous approaches [DBB92; BCS10], we present a DFT analysis based on probabilistic model checking. From a given DFT, a corresponding Markov automaton is automatically generated and analysed according to common measures such as the *reliability* or *mean-time-to-failure (MTTF)*. Using probabilistic model checking as a back-end guarantees precise analysis results and therefore prevents incorrect results. While existing works already used the same approach, we employ several optimisations during the state-space generation to mitigate the state-space explosion problem. These optimisations include *symmetry reduction*—a technique exploiting symmetric structures in the DFT—and *don't care* propagation—a technique which utilises the fact that during exploration certain components become irrelevant for the system failure.

Additionally, we present an approximation algorithm based on partial state-space generation. The approximation builds only parts of the state space and can be applied iteratively until the desired accuracy of the result is achieved. This algorithm guarantees lower and upper bounds on the analysis result while offering better scalability for larger models. We also present a formalisation of DFTs in *satisfiability modulo theories (SMT)* [KS16b] which is used as a pre-processing step in the approximation.

We implemented the DFT analysis in the tool Storm-dft as part of the Storm model checker [12]. Our evaluation shows that Storm-dft is state-of-the-art for DFT analysis.

1.2.3 Applications

Problem | Dynamic fault trees have been used to model (sub-)systems in a wide range of application areas. We refer to the FFORT benchmark collection [RBNS+19] for a list of prominent DFT models. All these models however have in common that their size is rather small: most DFTs have around 20 elements. Larger DFTs have only been constructed by artificially introducing redundancies [JGKR+17]. More detailed case studies are therefore required to properly evaluate the benefits and disadvantages of using DFTs.

Contribution | We apply DFT analysis for two case studies: a vehicle guidance system in the automotive domain and the infrastructure and corresponding routing options in railway station areas.

For the vehicle guidance system, we automatically generate a DFT for a given safety concept and a partitioning of the functionality on a corresponding E/E-architecture. Our analysis of the resulting DFTs via probabilistic model checking allows for more complex and insightful analysis queries.

For the railway station area and a given variety of trains and their corresponding routes in the station area, we are able to automatically derive the physical infrastructure components necessary for successful operation. We combine these routes with the fault trees for each infrastructure element in an automated approach and derive a complete DFT for the station area. Our analysis provides sensitivity measures to obtain the most critical switches for railway operations.

The DFTs created in both application areas are among the largest DFTs to date. Their analysis shows the feasibility and in particular scalability of our proposed DFT analysis. Additionally, our analysis results show that using DFTs instead of SFTs allows to create models which more accurately reflect reality.

1.3 Related Work

Throughout the thesis, we provide related works to each relevant aspect in the corresponding chapter. In the following, we give a short overview of other extensions of fault trees and general approaches for reliability analysis.

1.3.1 Other extensions of SFTs

Apart from DFTs, several other extensions of static fault trees have been proposed.

Boolean-logic driven Markov decision processes (BDMPs) | *BDMPs* [BB03] extend static fault trees by a so-called trigger mechanism. If the source of a trigger fails, the target is activated. Activation of a basic event changes the operation mode from passive to active and can for example increase the failure rate. Basic events in a BDMP can be modelled by Petri nets to accommodate more complex behaviour. The behaviour introduced by a trigger is very similar to the activation in a SPARE-gate in DFTs. Later extensions also incorporated dynamic gates such as the PAND-gate [Bou07]. While translations exist from BDMPs to DFTs [Bou07] and vice versa [10], the exact relationship between both formalisms and their expressiveness is still an open question. Recently, several of our contributions for DFTs have also been adopted for BDMPs, such as defining the semantics in terms of generalised stochastic Petri nets [KKB20] and analysis via partial state space generation [13].

Static/Dynamic fault trees (SD fault trees) | SD fault trees [BBHK⁺16] support certain dynamic parts within the static fault tree. Basic events can be modelled by so-called triggered continuous-time Markov chains which distinguish between two different modes. Upon failure of a static gate in the fault tree, a trigger can change the mode of associated basic event and therefore lead to increased failure rates. SD fault trees are a subclass of BDMPs [BBHK⁺16]. Compared to DFTs, SD fault trees lack expressiveness to model ordered failures or spare management. The restricted expressiveness however allows for efficient analysis similar to common SFT analysis [BBHK⁺16].

State/event fault trees (SEFTs) | SEFTs [KGF07] combine SFTs with finite state machines. States in a SEFT describe conditions which hold for some time while events occur instantaneously. SEFTs subsume both deterministic state machines and Markov chains and therefore are more expressive than DFTs. SEFTs are analysed by translation into deterministic and stochastic Petri nets [MC86]. However, efficient analysis support—as well as tool support—for SEFTs is lacking.

Pandora fault trees | Pandora fault trees [WP09] extend SFTs by three temporal gates: *priority-AND* (PAND), *simultaneous-AND* (SAND) and *priority-OR* (POR). Analysis is performed by using an extension of Boolean logic which also incorporates temporal laws. The three temporal gates are also present in DFTs and thus, DFTs subsume Pandora fault trees.

1.3.2 Other approaches for reliability analysis

Besides fault tree analysis, several other approaches are commonly used for reliability analysis. We refer to [RH03] and [TB17] for a general overview and further details.

Failure mode and effects analysis (FMEA) [IEC60812; Sta03] was one of the first structured approaches for reliability analysis. For each system component, the different failure modes and their effect on the system are listed in a table. Further information such as the probability of occurrence or the severity of a failure can also be recorded. FMEA is often used as a first step during reliability analysis and the identified failures form the basis for further models.

The Hazard and operability study (HAZOP) [Kle99] was initially developed in the chemistry sector and is similar to FMEA. A team identifies deviations from the intended behaviour by systematically going through a list of guide words—e.g. incorrect, more, less—and parameters—e.g. flow, temperature.

Reliability block diagrams (RBDs) [IEC61078; TB17] visualize the connections between different components which are required for successful operation. Each component is represented by a block. Blocks are either connected in series—if all blocks must be available for successful operation—or in parallel—when using redundancies.

The Architecture Analysis and Design Language (AADL) [AADL; BCKN+11] is an industry standard for modelling safety-critical systems. AADL models a systems by specifying its nominal behaviour, the error behaviour and fault injections describing how errors influence the nominal behaviour.

1.4 Outline

The thesis is structured in five parts. The main contributions are given in Parts II to IV.

Part I contains this introduction. The necessary preliminaries are given in Chapter 2. We start by introducing dynamic fault trees. We also define so-called failure traces—traces of failures in the DFT—which we use to define the behaviour of DFTs. Lastly, we introduce two models: Markov automata as the underlying model for DFT analysis and generalised stochastic Petri nets which are used to define the DFT semantics.

Semantics | In Part II we present the semantics for DFTs and in particular take a detailed look into the semantic intricacies occurring when combining DFT gates. We start by giving the precise semantics for each DFT gate in terms of GSPNs in Chapter 3. Chapter 4 contains a thorough investigation of multiple semantic issues occurring in DFTs. Besides describing these pitfalls, we also present our solutions to the issues and compare the proposed solutions to existing semantics.

Analysis | We present the analysis of DFTs via probabilistic model checking in Part III. The state-space generation including several optimisations is described in Chapter 5. In Chapter 6, we present an approximation technique based on abstraction refinement. Chapter 7 describes the implementation of the DFT analysis in the probabilistic model checker Storm. We evaluate Storm-dft on a set of DFT models from the FFORT benchmark suite and compare the performance to existing tools.

Applications | We present two industrial case studies of our DFT analysis toolset in Part IV. The first case study in Chapter 8 presents DFT models for vehicle guidance systems in the automotive domain. The second case study in Chapter 9 presents DFT models for train routings in a railway station area.

We conclude the thesis in Part V and provide an outlook into future work.

How to read the thesis | Readers familiar with DFTs, Markov automata or generalised stochastic Petri nets can skip the corresponding sections in the preliminaries as we are using standard notation. However, it might be good to read Section 2.2.4 as the concept of DFT traces is not commonly used in this form—even though it should be intuitive. The details of DFT traces are in particular relevant for Part II.

All three Parts II to IV can be read independent from one another. Both chapters in Part IV are also independent and can be read without previous knowledge.

Each chapter starts with a list of assumption about the structure of the considered DFTs. We also mention the corresponding publications the chapter is based upon. At the end of each chapter, we provide a single-page summary of the major messages and insights. For a quick overview, we therefore suggest to read the summary of each chapter and dive into the details as needed.

For convenience and quick reference, we provide an overview of our notation with links to the corresponding definitions in Appendix E on page 361. Additionally, we also provide an index in Appendix F on page 365. Boldfaced page numbers in the index reference the definition of a term. For easier navigation, we also highlight the name of newly defined terms in the page margin.

1.5 Personal Contributions

Following the regulations of the RWTH Aachen University, I shortly state my personal contributions to each of the relevant papers on which this thesis is based upon. I want to preface this list by stating my personal opinion that research is very much a team effort which yields the best results through the exchange and discussion of ideas. Ideas are constantly shaped, discussed, adapted, implemented, evaluated, improved—and sometimes also discarded. It is therefore hard to quantify the contributions of each author with respect to research results. Nevertheless, I give my—necessarily subjective—view on the contributions to each of the relevant papers in chronological order and according to the best of my recollection.

- M. Volk, S. Junges and J. Katoen. '*Advancing Dynamic Fault Tree Analysis - Get Succinct State Spaces Fast and Synthesise Failure Rates*'. *SAFECOMP*. Volume 9922. Lecture Notes in Computer Science. Springer, 2016, pages 253–265. DOI: 10.1007/978-3-319-45477-1_20
  This first paper on DFT analysis was initiated by Sebastian Junges as a continuation of his previous work [Jun15]. Seeing that the existing DFT analysis tools were not as fast as expected, the goal was to implement a simple, prototypical translation from DFTs to CTMCs. When I joined the group, a first, very rough version of a DFT analysis tool existed[1]. I significantly improved the implementation in close collaboration with Sebastian by adding model checking calls, several optimisations such as modularisation and don't cares, and improving the overall performance by extensive fine-tuning in C++. The initial draft of the resulting paper was created by Sebastian while I performed the evaluation. All three authors extended the draft and revised the paper.

- M. Volk, S. Junges and J. Katoen. '*Fast Dynamic Fault Tree Analysis by Model Checking Techniques*'. *IEEE Trans. Ind. Informatics* 14.1 (2018), pages 370–379. DOI: 10.1109/TII.2017.2710316
  Seeing that the optimisations for the DFT analysis were still not enough to mitigate the state space explosion problem, we thought about approximation techniques. To the best of my recollection, I came up with the initial idea of using a partial state space exploration which was significantly refined and improved during fruitful discussions with Sebastian Junges. I implemented the approach with input from Sebastian, integrated further improvements such as an adapted modularisation and also performed the evaluation. Sebastian created the SMT encoding of DFTs. The initial draft of the paper was mostly written by Sebastian. All three authors revised and improved the paper and its subsequent revision.

[1]Thanks to Git versioning the exact state of the implementation can be recalled: https://github.com/moves-rwth/storm/tree/1faa2c6f9e

- M. Ghadhab et al. '*Model-Based Safety Analysis for Vehicle Guidance Systems*'. *SAFECOMP*. Volume 10488. Lecture Notes in Computer Science. Springer, 2017, pages 3–19. DOI: 10.1007/978-3-319-66266-4_1
  During a collaboration with BMW AG, we aimed to apply the existing DFT analysis framework on a realistic case study. We developed a translation from function block diagrams to DFTs. Through frequent meetings, we revised the DFTs according to the feedback from BMW AG. I implemented the complete tool chain using our DFT tool as back-end and performed the experiments. Sebastian Junges and I drafted most parts of the paper. All authors extended and revised the paper.

- M. Ghadhab et al. '*Safety analysis for vehicle guidance systems with dynamic fault trees*'. *Reliab. Eng. Syst. Saf.* 186 (2019), pages 37–50. DOI: 10.1016/j.ress.2019.02.005
  The journal article is an extension of the previous conference paper. I improved some of the computation in the tool back-end and performed the detailed evaluation. Sebastian Junges and I drafted the first version by extending and revising the conference paper. All authors provided feedback and revised the final version of the journal paper.

- S. Junges et al. '*One Net Fits All - A Unifying Semantics of Dynamic Fault Trees Using GSPNs*'. *Petri Nets*. Volume 10877. Lecture Notes in Computer Science. Springer, 2018, pages 272–293. DOI: 10.1007/978-3-319-91268-4_14
  One issue while developing the DFT analysis tool and comparing to other tools was the fact that different semantics of DFTs are present in the literature. To alleviate this issue, we set out to develop a generalised semantics. The use of generalised stochastic Petri nets was proposed by Sebastian Junges. The templates for the DFT elements were conceived by both us. A first version of the templates was developed in a Master thesis under both our supervision. We revised the templates and I revised the implementation in our tool. All four authors discussed the different DFT semantics and their representation in the GSPN in great detail. Sebastian and I took the lead on writing the paper, and all authors revised the paper.

- M. Volk et al. '*A DFT Modeling Approach for Infrastructure Reliability Analysis of Railway Station Areas*'. *FMICS*. Volume 11687. Lecture Notes in Computer Science. Springer, 2019, pages 40–58. DOI: 10.1007/978-3-030-27008-7_3
  This paper is the result of a fruitful collaboration within the RTG 'UnRAVeL'. The aim was to apply the existing DFT framework in the railway domain. We had many interesting and fruitful discussions about railway operations in general and the applicability of DFTs. Through extensive feedback sessions, Norman Weik and I developed the general DFT models for the railway station areas. I implemented the tool-chain based on data pre-processed by Norman and I also performed the

evaluation. Norman and I took the lead on writing the paper. All authors provided feedback and revised the paper.

A complete list of my co-authored publications is given in Appendix B.2 on page 347.

Novel contributions | This thesis also contains new content not published before. Main novel contributions are the correctness proof of the GSPN translation throughout Chapter 3, the correctness proofs for the state-space generation algorithm and optimisations in Chapter 5 and a completely new evaluation of the tool Storm-dft in Chapter 7.

2 Preliminaries

We introduce some basic notation in Section 2.1. Fault trees are defined in detail in Section 2.2, in particular our constraints on DFTs, DFT traces and the state of a DFT. Note that the precise DFT semantics are given in Part II. We introduce Markov automata in Section 2.3 and generalised stochastic Petri nets in Section 2.4.

2.1 Basic Notation

Numbers | Let $\mathbb{N}$ denote the set of natural numbers without zero and $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ the set of natural numbers with zero. Further, $\mathbb{R}$ denotes the set of real numbers, $\mathbb{R}_{\geq 0} = \{r \in \mathbb{R} \mid r \geq 0\}$ the set of non-negative real numbers and $\mathbb{R}_{>0} = \mathbb{R}_{\geq 0} \setminus \{0\}$ the set of positive real numbers. If not further specified, intervals are over the real numbers and of the form $[a, b] \subseteq \mathbb{R}$.

Sets | For a set A, the power set of A is denoted by 2^A and the size of A is given by $|A|$. For two sets A, B with $A \cap B = \emptyset$, we denote the disjoint union of A and B by $A \uplus B$. The Cartesian product of two sets A, B is given by $A \times B = \{(a, b) \mid a \in A, b \in B\}$.

Sequences | Sequences are concatenations of elements from a set.

> **Definition 2.1 (Sequence)** Let K be a finite set. A (finite) *sequence* ξ over K is given by $\xi = k_1 k_2 \ldots k_n$ with $k_i \in K$ for all $1 \leq i \leq n$. The set of all *sequences* over K is denoted by K^*. sequence sequences

We use ξ_i to denote the i-th item of a sequence ξ and we use $k \in \xi$ as shorthand for $\exists i \in \mathbb{N} : \xi_i = k$. We use $\xi' \subseteq \xi$ to indicate that sequence $\xi' = k_1 k_2 \ldots k_m$ is a *prefix* of sequence $\xi = k_1 k_2 \ldots k_m k_{m+1} \ldots k_n$, $n \geq m$. The *size* (or length) of a finite sequence $\xi = k_1 \ldots k_n$ is given by $|\xi| = n$. The *empty sequence* is denoted by ε and has size $|\varepsilon| = 0$. ·prefix ·size ·empty sequence

Relations | A (binary) relation R is a set $R \subseteq A \times B$. If $A = B$, we call R a relation on A. In the remainder, we use aRb to indicate that $(a, b) \in R$.

> **Definition 2.2 (Strict partial order)** Let $\prec$ be a relation on A. Relation $\prec$ is called
> - irreflexive if $\neg(a \prec a)$ for all $a \in A$.
> - asymmetric if $a \prec b \implies \neg(b \prec a)$ for all $a, b \in A$, and

- transitive if $a \prec b \land b \prec c \implies a \prec c$ for all $a, b, c \in A$.

·strict

Relation $\prec$ is a *strict partial order* if it is irreflexive, asymmetric and transitive.

Graphs | A (directed) graph is a tuple $G = (V, E)$ consisting of a set of vertices V and an edge relation $E \subseteq V \times V$. The graph is undirected if for all $(v, v') \in E$ also $(v', v) \in E$ holds. A path in graph G is a sequence of vertices $v_1 \dots v_n$ such that $v_i \in V$ for all $1 \leq i \leq n$ and $(v_j, v_{j+1}) \in E$ for all $1 \leq j < n$. We write $v_1 \dots v_n \in G$ to indicate that $v_1 \dots v_n$ is a path in G.

2.1.1 Probability theory

We introduce some concepts from probability theory and base our definitions on [Tim13]. For a more in-depth introduction, we refer to [Tim13; BK08; Ste09].

Random variable | A *random variable* is a function which maps outcomes from an experiment to values. For a random variable X, we use $\mathbf{P}(X \in E)$ to denote the probability that the value of X is in a given set E. The range of the random variable is called the *sample space* and we distinguish between discrete and continuous ranges, i.e. discrete and continuous random variables, respectively.

Probability distribution | We use *probability distributions* to specify the probabilities for the outcomes of a random variable.

probability distribution

Definition 2.3 (Probability distribution) Let $\omega = (type, parameter^*)$ be a *probability distribution* where

- *type* defines the type of probability distribution, and
- *parameter** is a list of parameters describing the shape of the distribution.

We use Ω to denote the set of probability distributions.

·zero

For easier notation, we introduce the *zero distribution* $\omega = (zero, \varepsilon)$ which always returns probability 0. Strictly speaking it is therefore not a probability distribution.

2.1.1.1 Discrete probability distribution

·discrete

Definition 2.4 (Discrete probability distribution) Let S be a countable sample space. A *discrete probability distribution* is a function

$$\mu : S \to [0, 1] \text{ with } \sum_{s \in S} \mu(s) = 1.$$

The set of a discrete probability distributions over S is denoted by Distr(S).

Let X be a random variable over sample space S described by the discrete probability distribution μ. Then the occurrence of outcome $s \in S$ has probability $\mathbf{P}(X = s) = \mu(s)$.

Bernoulli distribution | One example discrete probability distribution is the *Bernoulli distribution* [Ber13] with $\omega = (\mathit{Bernoulli}, p)$ over a binary sample space $S = \{0, 1\}$. The Bernoulli distribution yields outcome 1 with probability p and outcome 0 with probability $1-p$, i.e. $\mathbf{P}(X = 1) = p$ and $\mathbf{P}(X = 0) = 1-p$. ·Bernoulli

2.1.1.2 Continuous probability distribution

In the continuous case, we use $\mathbb{R}$ as the sample space. We assume that the probability of each individual outcome is zero. Instead, we assign a probability to each interval $[a, b]$.

Definition 2.5 (Probability density function) Let X be a continuous random variable over sample space $\mathbb{R}$. The *probability density function* of X is a function $f(x) : \mathbb{R} \mapsto \mathbb{R}_{\geq 0}$ with $\int_{-\infty}^{\infty} f(x)\,\mathrm{d}x = 1$ such that

probability density function

$$\mathbf{P}(X \in [a, b]) = \int_a^b f(x)\,\mathrm{d}x.$$

Definition 2.6 (Cumulative distribution function) The *cumulative distribution function* for random variable X with probability density function f is given by

cumulative distribution function

$$F(x) = \mathbf{P}(X < x) = \int_{-\infty}^{x} f(y)\,\mathrm{d}y.$$

Definition 2.7 (Expected value) The *expected value* of a random variable X with probability density function f is given by

expected value

$$\mathbf{E}(X) = \int_{-\infty}^{\infty} x \cdot f(x)\,\mathrm{d}x.$$

Note that the expected value does not always exist.

Exponential distribution | In this work, we focus on exponential distributions.

Definition 2.8 (Exponential distribution) An *exponential distribution* $\omega = (\mathit{exp}, \lambda)$ with *rate* $\lambda > 0$ has the following probability density function $f(x)$ and cumulative distribution function $F(x)$:

·exponential
rate

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & \text{if } x > 0 \\ 0 & \text{otherwise,} \end{cases} \qquad F(x) = \begin{cases} 1 - e^{-\lambda x} & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

The expected value of an exponential distribution is the reciprocal of its rate:

$$\mathbf{E}(X) = \int_{-\infty}^{\infty} x \cdot \lambda e^{-\lambda x}\,\mathrm{d}x = \frac{1}{\lambda}.$$

If it is clear from context that the considered distribution is an exponential distribution, we use the shorthand $\omega = \lambda$. The set of exponential distributions is denoted by $\Omega_{exp} \subset \Omega$. In the special case $\lambda = 0$, we obtain the *zero distribution*.

We refer to [TB17, Chapter 3] for an overview of other common probability distributions in the context of reliability engineering.

2.2 Fault Trees

We first informally introduce DFTs and in particular the failure behaviour of all gates in Section 2.2.1. Afterwards, we formally define DFTs in detail in Section 2.2.2 and introduce the notion of well-formed and conventional DFTs in Section 2.2.3. In Section 2.2.4, we introduce failure traces to formalise the behaviour of a DFT in terms of the failure of elements. Lastly, we define the state of a DFT after a given failure trace in Section 2.2.5.

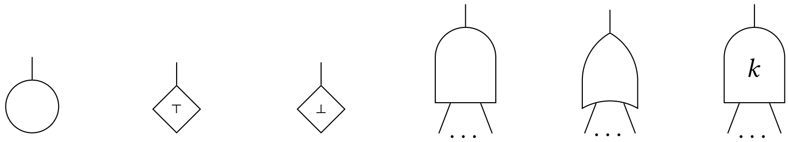
2.2.1 Introducing Dynamic Fault Trees

A *dynamic fault tree (DFT)* is a directed acyclic graph. The type of each node represents the type of *DFT element* and the outgoing edges represent the *children* of an element—also called inputs. We informally introduce all DFT elements in the following. The corresponding graphical symbols are provided in Figure 2.1. The precise semantics of DFTs are discussed in Part II.

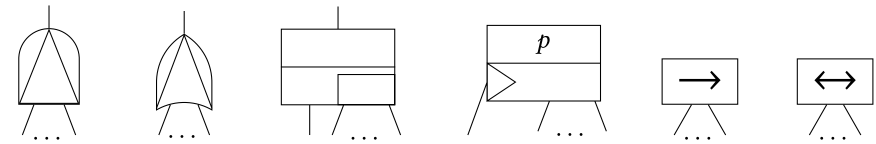
Basic events (BEs) | Nodes without children are called *basic events (BEs)*. Each BE represents a component of the real-life system. We distinguish between two types of BEs: stochastic events and constant events. A *stochastic event (SE)* (Figure 2.1a) fails according to its associated failure distribution. Constant events are either *constant failed* ($\mathrm{CONST}_{\top}$) (Figure 2.1b) or *constant fail-safe* ($\mathrm{CONST}_{\perp}$) (Figure 2.1c).

Static gates | Nodes with children are called *gates* and divided into *static* and *dynamic* gates. We start with the failure behaviour of static gates.

- The *AND*-gate (Figure 2.1d) fails if all its children have failed.
- The *OR*-gate (Figure 2.1e) fails if at least one of its children has failed.
- The *voting*-gate (VOT_k) (Figure 2.1f) with threshold k is a generalisation of both former gates and fails if at least k of its children have failed.



(a) SE (b) $\text{CONST}_\top$ (c) $\text{CONST}_\bot$ (d) AND (e) OR (f) VOT_k



(g) PAND (h) POR (i) SPARE (j) PDEP_p (k) SEQ (l) MUTEX

Figure 2.1: Overview of DFT elements

Dynamic gates | In dynamic gates, the order of the failures of the children is relevant.

- The *priority AND*-gate (PAND) (Figure 2.1g) fails if all children fail in order from left to right. If one of the children fails out of order, the PAND becomes fail-safe.
- Similarly, the *priority OR*-gate (POR) (Figure 2.1h) fails if the first child fails before all other children.
- The *SPARE*-gate (Figure 2.1i) models spare management. Initially, the first child—the *primary child*—is *claimed* by the SPARE and used. The other children are not actively used and therefore might have a reduced failure probability. If the first child fails, the next available child—in order from left to right—is claimed. A child is *available for claiming* if it is operational and not already in use by another SPARE. A claimed child is also activated and might have a higher failure probability than before. The SPARE-gate fails if all claimed children have failed and no child is available for claiming any more.
- The *probabilistic dependency* (PDEP_p) (Figure 2.1j) forwards the failure of the first child—the *trigger*—to the other children—the *dependent elements*—with probability p. The PDEP_p does not fail itself.
- The *sequence enforcer* (SEQ) (Figure 2.1k) ensures the children are failing in order from left to right. Failures out of order are not possible. The SEQ does not fail.
- The *mutual exclusion* (MUTEX) (Figure 2.1l) ensures that only one of its children fails. More than one child cannot fail. The MUTEX does not fail.

2.2.2 Dynamic Fault Trees

We formally define DFTs in the following. Our notation is largely based on [Jun15].

static gate types

Definition 2.9 (Gate types) The set of *static gate types* is given by

$$\text{StaticGates} = \{\text{AND}, \text{OR}\} \cup \{\text{VOT}_k \mid k \in \mathbb{N}\}\,.$$

dynamic gate types

The set of *dynamic gate types* is given by

$$\text{DynamicGates} = \{\text{PAND}, \text{POR}, \text{SPARE}, \text{SEQ}, \text{MUTEX}\} \cup \left\{\text{PDEP}_p \mid p \in [0,1]\right\}.$$

functional dependency

We use VOT to denote the set of all voting gates $\text{VOT} := \{\text{VOT}_k \mid k \in \mathbb{N}\}$. If necessary, we distinguish between inclusive $\text{PAND}_{\leq}$ and exclusive $\text{PAND}_{<}$. Similarly, we distinguish between $\text{POR}_{\leq}$ and $\text{POR}_{<}$. The *functional dependency* $\text{FDEP} := \text{PDEP}_1$ is a special case of the probabilistic dependency. PDEP denotes the set of all probabilistic dependencies $\text{PDEP} := \left\{\text{PDEP}_p \mid p \in [0,1]\right\}$ and also incorporates FDEP.

·gate types

Definition 2.10 (DFT element types) The set of DFT *gate types* is given by

$$\text{Gates} = \text{StaticGates} \uplus \text{DynamicGates}.$$

·leaf types

The set of DFT *leaf types* is given by

$$\text{Leaves} = \{\text{CONST}_{\top}, \text{CONST}_{\bot}, \text{SE}\}\,.$$

The set of all DFT element types is given by

$$\text{Elements} = \text{Leaves} \uplus \text{Gates}.$$

An event of any type in Leaves is also called a *basic event (BE)*.

·constant failed
·constant fail-safe

We have two types of constant BEs: the *constant failed* event $\text{CONST}_{\top}$ and the *constant fail-safe* event $\text{CONST}_{\bot}$.

stochastic event

SE denotes a *stochastic event* with an arbitrary probability distribution.

DFT

Definition 2.11 (Dynamic Fault Tree (DFT)) A *Dynamic fault tree (DFT)* over the set of probability distributions Ω is a tuple $\mathcal{F}^{\Omega} = (V, \sigma, \mathit{Type}, \mathit{top}, \Theta)$, where

·nodes

- V is a finite set of *nodes*.

·children

- $\sigma : V \to V^*$ defines the sequence of (ordered) *children* of a node.
- $\mathit{Type} : V \to \text{Elements}$ defines the element type of a node.

·top-level event (TLE)

- $\mathit{top} \in V$ is the *top-level event (TLE)*.

- $\Theta : \{v \in V \mid Type(v) = \mathsf{SE}\} \to (\Omega, \Omega)$ is the attachment function mapping an element of type SE to two *failure distributions*. For a SE v with $\Theta(v) = \left(FD_a, FD_p\right)$, the first probability distribution FD_a represents the failure behaviour while the SE is *actively* used and the second probability distribution FD_p represents the failure behaviour in *passive* mode.

failure distribution
·active
·passive

2

We drop the superscript Ω if it is clear from context—or not relevant.

The nodes V are also often called *elements* of the DFT and the children $\sigma(v)$ of a DFT node are often called *inputs*. Using the notation from sequences (cf. Definition 2.1), $\sigma(v)_i$ denotes the i-th child of v and $v' \in \sigma(v)$ is shorthand for $\exists i \in \mathbb{N} : \sigma(v)_i = v'$.

·elements
·inputs

We use $\rho(v) := \{v' \in V \mid v \in \sigma(v')\}$ to denote the parents of a node. We use $\rho^K(v) = \{v' \in \rho(v) \mid Type(v') = K\}$ to restrict the parents to DFT elements of type K.

For a node $v \in V$ we also write $v \in \mathcal{F}$. For an element type $K \in \mathsf{Elements}$ we define the set of all K by $K_{\mathcal{F}} := \{v \in V \mid Type(v) = K\}$. For a node v and an element type K we say 'v is a K' if $v \in K_{\mathcal{F}}$. For a set of element types $L \subseteq \mathsf{Elements}$, we analogously define $L_{\mathcal{F}} := \{v \in V \mid Type(v) \in L\}$ and say 'v is an L' if $v \in L_{\mathcal{F}}$.

In general, we often omit the subscript $\mathcal{F}$ if the DFT is clear from context.

Basic events | Stochastic events SE can be attached with arbitrary probability distributions. We use SE_{type} to denote SE with probability distribution *type*, i.e.

$$Type(v) = \mathsf{SE}_{type} \iff v \in \mathsf{SE}_{\mathcal{F}} : \Theta(v) \in \left(\Omega_{type}, \Omega_{type}\right).$$

For instance, SE_{exp} denotes all stochastic events with exponential failure distributions. For the two failure distributions, we sometimes explicitly add the DFT element as a superscript for clarity: $\Theta(v) = \left(FD_a^v, FD_p^v\right)$.

The set of all Leaves is also called the set of *basic events (BEs)* and is defined as $\mathsf{BE}_{\mathcal{F}} := \{v \in V \mid Type(v) \in \mathsf{Leaves}\}$. A node $v \in \mathsf{BE}_{\mathcal{F}}$ is called a *basic event (BE)*.

basic events (BEs)
BE

Graph of a DFT | DFTs induce a graph structure and also a notion of reachability.

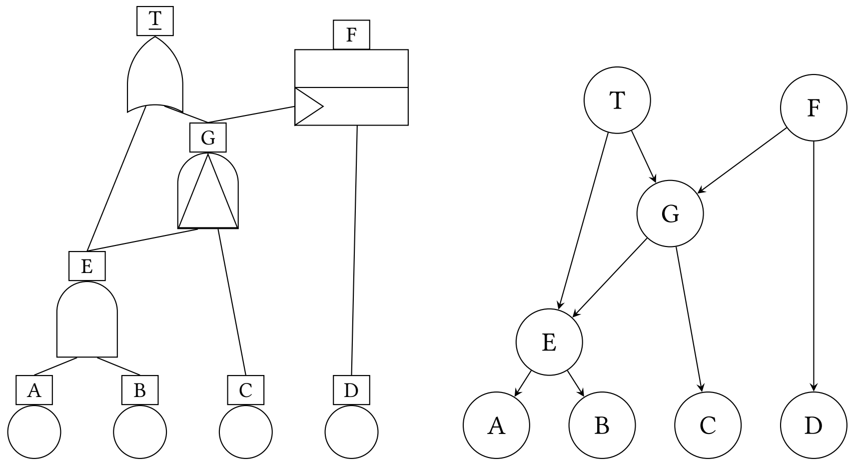
Definition 2.12 (Graph of a DFT) Let $\mathcal{F} = (V, \sigma, Type, top, \Theta)$ be a DFT. The *graph* of $\mathcal{F}$ is given by $G(\mathcal{F}) = (V, E(\sigma))$, where the edge relation induced by σ is defined as $E(\sigma) = \{(v, v') \in V \times V \mid v' \in \sigma(v)\}$. The undirected graph of a DFT is given by $G^{\rightleftharpoons}(\mathcal{F}) = \left(V, E(\sigma) \cup E(\sigma)^{-1}\right)$

·graph

Definition 2.13 (Reachable DFT elements) Let $\mathcal{F}$ be a DFT and $v \in \mathcal{F}$ a DFT element. The DFT elements *reachable* from v in the graph $G(\mathcal{F})$ are given by

·reachable

$$\mathrm{reach}(\mathcal{F}, v) = \{v' \in \mathcal{F} \mid \exists \text{ path } v v_1 \ldots v_n v' \text{ in } G(\mathcal{F})\}.$$



(a) Example DFT $\mathcal{F}$ (b) Graph of $\mathcal{F}$

Figure 2.2: Examples of DFT and its graph

The set of DFT elements (downwards) reachable from v in the undirected graph $G^{\rightleftharpoons}(\mathcal{F})$ is given by

$$\begin{aligned}\mathsf{reachExtended}(\mathcal{F}, v) = \{v' \in \mathcal{F} \mid \exists \text{ path } v_1 \ldots v_n v' \text{ in } G^{\rightleftharpoons}(\mathcal{F})\\ \text{such that } v_1 \in \sigma(v) \text{ and } v \neq v_i \text{ for all } 1 \leq i \leq n\}.\end{aligned}$$

Subtree of DFT | We use the reachability notion to define subtrees in the DFT.

Definition 2.14 (Subtree) Let $\mathcal{F}$ be a DFT and $v \in \mathcal{F}$ be a DFT element. The *subtree* with top element v is given by $\mathsf{subtree}(\mathcal{F}, v) = (V', \sigma', \mathit{Type}', v, \Theta')$ with

subtree

- $V' = \mathsf{reach}(\mathcal{F}, v)$ and
- for all $v \in V'$ we have $\sigma'(v) = \sigma(v)$, $\mathit{Type}'(v) = \mathit{Type}(v)$ and $\Theta'(v) = \Theta(v)$.

·extended subtree

The *extended subtree* with top element v is given by $\mathsf{subtreeExtended}(\mathcal{F}, v)$ which is the DFT restricted to $V = \mathsf{reachExtended}(\mathcal{F}, v)$.

·real subtree

We call $\mathsf{subtreeExtended}(\mathcal{F}, v)$ a *real subtree*, if the parents of v are not part of the extended subtree, i.e. $\mathsf{reachExtended}(\mathcal{F}, v) \cap \rho(v) = \emptyset$.

In the following, we often identify a subtree by its top element v.

Example 2.1 (Graph and subtree of DFT) We consider the DFT $\mathcal{F}$ in Figure 2.2a. The corresponding graph $G(\mathcal{F})$ is depicted in Figure 2.2b.

The elements reachable from T in $\mathcal{F}$ are all elements except F and D. The subtree for T, $\mathsf{subtree}(\mathcal{F}, T)$, can therefore be obtained from $\mathcal{F}$ by simply removing F and D. The subtree for E is given by the elements E, A and B and the subtree for G is given by the elements A, B, C, E and G.
The set of extended reachable elements from T is $\mathsf{reachExtended}(\mathcal{F}, T) = V_{\mathcal{F}}$. The extended subtree for T is $\mathcal{F}$ and it is a real subtree. The extended subtree for E is the same as the (unextended) subtree and it also is a real subtree. The extended subtree for G incorporates T as well, because the path $E\,T$ reaches T from a child of G in the undirected graph $G^{\rightleftharpoons}(\mathcal{F})$. As the parent T of the top element G is also contained in the extended subtree, this tree is not a real subtree.

2.2.3 Constraints on DFTs

We require certain constraints to hold on DFTs. Most of these constraints were introduced in [Jun15, Definition 4.9].

2.2.3.1 Well-formedness constraints

The first constraints ensure that the DFT is well-formed.

DFT Constraint 1 (DFT is acyclic) The graph $G(\mathcal{F})$ of a DFT $\mathcal{F}$ is *acyclic*.

DFT Constraint 2 (Exactly the BEs have no children) A node $v \in \mathcal{F}$ has no children iff v is a BE.

$$\forall v \in \mathcal{F} : \sigma(v) = \varepsilon \iff v \in \mathsf{BE}_{\mathcal{F}}.$$

DFT Constraint 3 (Valid threshold for VOT) For VOT_k with n children, it must hold $1 \leq k \leq n$.

$$\forall v \in V : \mathit{Type}(v) = \mathsf{VOT}_k \implies 1 \leq k \leq |\sigma(v)|.$$

DFT Constraint 4 (TLE is neither SEQ nor PDEP) The top-level event top is neither a SEQ nor a PDEP.

$$\mathit{Type}(\mathit{top}) \notin \{\mathsf{SEQ}, \mathsf{PDEP}\}.$$

DFT Constraint 5 (No parents for SEQ and PDEP) SEQ and PDEP have no ingoing edges.

$$\forall v \in V : \mathit{Type}(v) \in \{\text{SEQ}, \text{PDEP}\} \implies \rho(v) = \emptyset.$$

DFT Constraint 6 (SEQ and PDEP are at least binary) SEQ and PDEP have at least two children.

$$\forall v \in V : \mathit{Type}(v) \in \{\text{SEQ}, \text{PDEP}\} \implies |\sigma(v)| \geq 2.$$

DFT Constraints 3 to 6 are not strictly necessary and could be omitted. However, without these constraints we would need to define the behaviour of the DFT in the excluded corner cases.

·well-formed

Definition 2.15 (Well-formed DFT) DFT $\mathcal{F}$ is *well-formed* iff DFT Constraints 1 to 6 are satisfied.

Remark 2.1 In this thesis, we assume that all DFTs are well-formed.

In the following, we introduce additional constraints which are often used to make modelling and analysis of DFTs easier. Moreover, some of the constraints prevent special cases where the behaviour of the DFT is not clear any more.

2.2.3.2 Restriction to exponential distributions

The first constraint restricts the failure distributions to exponential distributions. Exponential distributions are easier to handle and are supported by a broad range of models, e.g. Markov automata (cf. Section 2.3) and generalised stochastic Petri nets (cf. Section 2.4). Most DFT analysis approaches and existing tools restrict themselves to exponential distributions, cf. e.g. [DBB92; Rai05; RS15] and also Section 4.3.2.

DFT Constraint 7 (Restriction to exponential distributions) The *failure distributions* in DFT $\mathcal{F}^{\Omega}$ are restricted to *exponential distributions*, i.e. $\mathcal{F}^{\Omega_{exp}}$. The attachment function Θ therefore has the form $\Theta : \{v \in \text{SE}\} \to \left(\Omega_{exp}, \Omega_{exp}\right)$. For $v \in \text{SE}$ we use $\Theta(v) = \left(\lambda_a, \lambda_p\right)$ to denote the failure distributions with *failure rates* λ_a and λ_p. The *active* failure rate is given by λ_a and the *passive* failure rate is given by λ_p.

failure rate

·dormancy factor

The passive failure rate is often given by the *dormancy factor* d. The dormancy factor with $0 \leq d \leq 1$ defines the ratio between passive and active failure rate $d = \lambda_p/\lambda_a$. The passive failure rate can therefore also be expressed as $\lambda_p = d \cdot \lambda_a$.

2.2.3.3 Spare modules

The next constraints ensure that the claiming in a SPARE gate is distinctly defined. We introduce so-called spare modules consisting of all DFT elements which are claimed and activated when a SPARE-gate switches to another spare component.

Before we define spare modules, we first define module paths as a helper function. A module path is a path which does not contain FDEPs or SEQs and does not connect a SPARE with any of its children—but only with its parents.

Definition 2.16 (Module path) Let $\mathcal{F}$ be a DFT. A path $v_0 v_1 \dots v_n$ in the undirected graph $G^{\rightleftharpoons}(\mathcal{F})$ is as *module path* if

·module path

- $v_i \notin \mathsf{PDEP}$ and $v_i \notin \mathsf{SEQ}$ for all $0 \le i \le n$,
- $v_i \in \mathsf{SPARE} \implies v_{i+1} \in \rho(v_i)$ for all $0 \le i \le n{-}1$, and
- $v_i \in \mathsf{SPARE} \implies v_{i-1} \in \rho(v_i)$ for all $1 \le i \le n$

A module is uniquely identified by it module representative. A representative for a module is either a child of a SPARE or the top-level element.

Definition 2.17 (Module representatives) Let $\mathcal{F}$ be a DFT. The set of *spare module representatives* for $\mathcal{F}$ is given by

·spare module representative

$$\mathsf{SpareModuleRepr}_{\mathcal{F}} = \bigcup_{s \in \mathsf{SPARE}_{\mathcal{F}}} \sigma(s).$$

The set of *module representatives* is given by

·module representative

$$\mathsf{ModuleRepr}_{\mathcal{F}} = \mathsf{SpareModuleRepr}_{\mathcal{F}} \cup \{top\}\,.$$

All elements which are connected to a module representative by a module path belong to the same module. All elements which do not belong to any spare module, belong to the top module of the top-level element.

Definition 2.18 (Module) Let $\mathcal{F}$ be a DFT. The *module* for spare module representative $r \in \mathsf{SpareModuleRepr}_{\mathcal{F}}$ is given by

·module

$$\mathsf{Module}(r) = \left\{ v \in \mathcal{F} \mid \text{there exists a module path } p = v v_1 \dots v_n r \text{ in } G^{\rightleftharpoons}(\mathcal{F}) \right\}.$$

The *top module* for module representative *top* is given by

·top module

$$\begin{aligned} \mathsf{Module}(top) = &\left\{ v \in \mathcal{F} \mid \text{there exists a module path } p = v v_1 \dots v_n top \text{ in } G^{\rightleftharpoons}(\mathcal{F}) \right\} \\ &\cup \left\{ v \in V \mid v \notin \mathsf{Module}(r) \text{ for all } r \in \mathsf{SpareModuleRepr}_{\mathcal{F}} \right\}. \end{aligned}$$

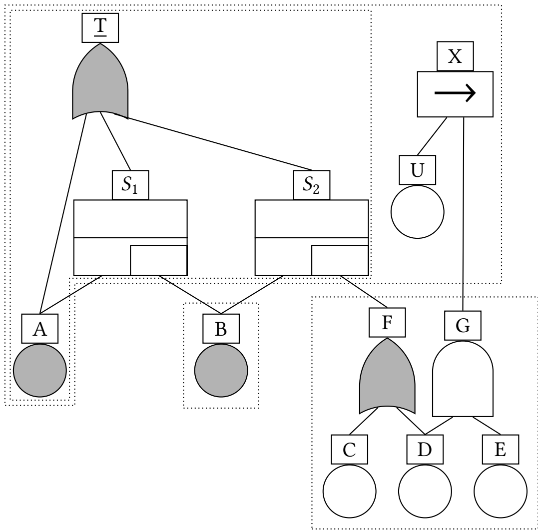


Figure 2.3: Modules (boxes) and module representatives (grey)

·primary module
·spare module

Given a SPARE s, the *primary module* is $\text{Module}(\sigma(s)_1)$ of the first child. For the other children $i > 1$, $\text{Module}(\sigma(s)_i)$ is called a *spare module*. Note that not all elements in the top module necessarily have a module path to *top*. For example, elements which are children of PDEPs might not have such a connection.

DFT Constraint 8 (Independence of modules) The modules of a DFT $\mathcal{F}$ must be independent, i.e., disjoint.

$$\forall r, r' \in \text{ModuleRepr}_{\mathcal{F}} \text{ with } r \neq r' : \text{Module}(r) \cap \text{Module}(r') = \emptyset.$$

The modules must be independent subtrees. This constraint makes claiming and activation in a SPARE easier, because each module contains exactly one module representative. This representative is the child of a SPARE—or the TLE. Thus, claiming a child of a SPARE uniquely defines the complete module which is claimed. As modules cannot 'overlap', multiple SPAREs claiming the same module is also not possible.

Example 2.2 (Modules) We depict the modules of an example DFT in Figure 2.3. The module representatives are indicated by a grey background colour and the modules are given by the dotted boxes.
The module of spare representative B contains only this element $\text{Module}(B) = \{B\}$.

The module is a primary module of S_2 and also a spare module of S_1.
The module of representative F is $\mathsf{Module}(F) = \{C, D, E, F, G\}$. Note that G and E are also part of the module even though they are not part of the subtree of F. X is excluded from this module, because SEQs are not part of a module path.
The module of representative A also contains the TLE T which is a module representative itself. Thus, the modules of A and T are not independent. Note that X and U are not part of $\mathsf{Module}(A)$, because they are not reachable by a module path. However, both elements are part of the top module.

DFT Constraint 8 (Independence of modules) is sometimes lifted for the top module and primary modules. In those cases, primary modules can contain elements of the top module as well. As both the top module and primary modules are activated from the beginning, the 'overlap' makes no difference with respect to activation. Moreover, claiming is also not affected, because the TLE cannot be claimed.

The following two constraints ensure that the initial claiming of SPAREs is successful, i.e. each SPARE can claim its primary module.

DFT Constraint 9 (No constant failed events in primary modules) All primary modules of SPAREs do not contain $\mathsf{CONST}_\top$.

$$\forall v \in \mathsf{SPARE}_{\mathcal{F}} : \mathsf{Module}(\sigma(v)_1) \cap \mathsf{CONST}_\top = \emptyset.$$

DFT Constraint 10 (No sharing of primary modules) Primary modules of SPAREs cannot be shared, i.e. they belong to exactly one SPARE.

$$\forall v \in \mathsf{SPARE}_{\mathcal{F}} : \rho(\sigma(v)_1) = \{v\}\,.$$

The next constraint avoids having to formalize what happens when gates fail due to outside triggers. This is in particular useful for SPAREs where claiming and activation of remaining available children would need to be specified.

DFT Constraint 11 (Dependent events of PDEPs are BEs) PDEPs have only BEs for dependent events. The trigger can be of arbitrary type.

$$\forall v \in \mathsf{PDEP}_{\mathcal{F}} : \bigwedge_{i=2}^{|\sigma(v)|} \mathit{Type}(\sigma(v)_i) = \mathsf{BE}.$$

We ensure that only one constant failed element exists in a DFT. That way, the propagation of immediate failures can always be traced back to the unique $\mathsf{CONST}_\top$.

DFT Constraint 12 (Unique CONST$_\top$) There is at most one CONST$_\top$ element.

$$|\mathsf{CONST}_\top| \leq 1.$$

Lastly, we present two constraints for SEQs which allow to pre-compute the DFT elements which are allowed to fail.

DFT Constraint 13 (Inputs of SEQs are SEs) SEQ has only SEs as inputs.

$$\forall v \in \mathsf{SEQ}_{\mathcal{F}} : \bigwedge_{i=1}^{|\sigma(v)|} \mathit{Type}(\sigma(v)_i) = \mathsf{SE}.$$

DFT Constraint 14 (No dependencies to inputs of SEQ) The children of SEQs are not dependent events of PDEPs.

$$\forall s \in \mathsf{SEQ}_{\mathcal{F}} : \forall f \in \mathsf{PDEP}_{\mathcal{F}} : \bigwedge_{i=1}^{|\sigma(s)|} \bigwedge_{j=2}^{|\sigma(f)|} \sigma(s)_i \neq \sigma(f)_j.$$

We conclude by defining conventional DFTs which is the restricted class of DFTs we usually consider throughout this thesis. Note that the last two constraints are not part of conventional DFTs.

·conventional

Definition 2.19 (Conventional DFT) A DFT $\mathcal{F}$ is *conventional* iff it is *well-formed* (cf. Definition 2.15) and additionally satisfies the following constraints:

- DFT Constraint 7 (Restriction to exponential distributions),
- DFT Constraint 8 (Independence of modules),
- DFT Constraint 9 (No constant failed events in primary modules),
- DFT Constraint 10 (No sharing of primary modules),
- DFT Constraint 11 (Dependent events of PDEPs are BEs) and
- DFT Constraint 12 (Unique CONST$_\top$).

Conventional DFT are restricted enough such that for example claiming in SPAREs and triggering through PDEPs is intuitively clear. On the other side, conventional DFT have as few restrictions as possible to allow maximum flexibility when modelling and to encompass most common DFT semantics from the literature.

2.2.4 DFT Traces

In a DFT, different kinds of events can take place—occurrences of failures, claiming of spares, activation, etc. The events occurring in a DFT represent its behaviour. In the following, we will define the behaviour of a DFT in terms of valid traces over the DFT events. We start by introducing the possible events in a DFT.

Definition 2.20 (DFT events) Let $\mathcal{F}$ be a DFT. We distinguish the following events in a DFT:

- The *failure* of DFT node v is denoted by f_v. If $v \in \mathrm{SE}_{\mathcal{F}}$ fails by itself—and not due to a dependency—we call this an *SE failure* and indicate it by $\mathbf{f}_v$. failure ·SE failure
- Event cl_v^s indicates that SPARE s *claims* its child $v \in \sigma(s)$. ·claiming
- The *activation* of DFT node v is denoted by $\uparrow_v$. ·activation
- Event $\lightning_v$ indicates that the restriction of SEQ v is *violated*. ·violation
- We use $\oslash_v$ to denote that *no event* takes place for DFT node v. ·no event

The set of *DFT events* in $\mathcal{F}$ is given by ·DFT events

$$\mathsf{Events}_{\mathcal{F}} = \{f_v, \uparrow_v, \oslash_v \mid v \in \mathcal{F}\} \cup \left\{\mathrm{cl}_v^s \mid s \in \mathrm{SPARE}_{\mathcal{F}}, v \in \sigma(s)\right\} \cup \{\lightning_s \mid s \in \mathrm{SEQ}_{\mathcal{F}}\}.$$

We denote the set of *SE failures* in $\mathcal{F}$ by ·SE failures

$$\mathsf{Fails}_{\mathrm{SE}_{\mathcal{F}}} = \{\mathbf{f}_v \mid v \in \mathrm{SE}_{\mathcal{F}}\}.$$

The set of *no events* is given by ·no events

$$\varnothing_{\mathcal{F}} = \{\oslash_v \mid v \in \mathcal{F}\}.$$

We often omit the subscript $\mathcal{F}$ if the DFT is clear from the context.

The possible behaviour of a DFT is specified by traces over DFT events.

Definition 2.21 (Event traces) Let $\mathcal{F}$ be a DFT. An *event trace* τ in $\mathcal{F}$ is a sequence of DFT events ·event trace

$$\tau = e_1 e_2 \dots e_n \in \mathsf{Events}_{\mathcal{F}}^*,$$

such that all events except 'no event' $\oslash_v$ occur at most once in τ. In other words:

$$\forall e_i, e_j \in \tau, i \neq j : e_i \neq e_j \vee e_i \in \varnothing_{\mathcal{F}} \vee e_j \in \varnothing_{\mathcal{F}}.$$

The set of event traces in a DFT $\mathcal{F}$ is denoted by $\mathsf{Events}_{\mathcal{F}}^{\blacktriangleright} \subseteq \mathsf{Events}_{\mathcal{F}}^*$.

The most important events in a DFT are the SE failures. Each SE failure represents the failure of an external component which failed on its own. Traces over SE failures therefore represent the possible external failures which can occur in a system.

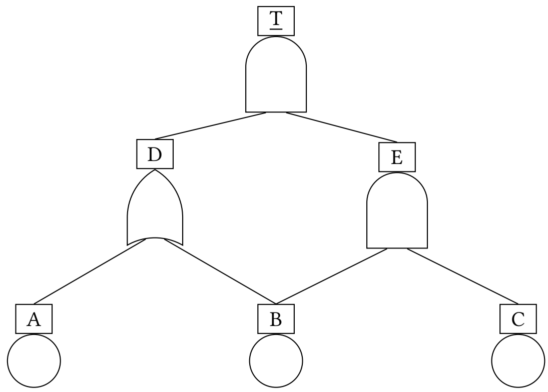


Figure 2.4: Example DFT for traces

·failure trace

Definition 2.22 (Failure traces) Let $\mathcal{F}$ be a DFT. A *failure trace* τ_{f}—also called an *ftrace*—in $\mathcal{F}$ is an event trace

$$\tau_{\mathsf{f}} = \mathsf{f}_{v_1}\ \mathsf{f}_{v_2}\ \ldots\ \mathsf{f}_{v_n} \in {\mathsf{Fails}_{\mathsf{SE}_{\mathcal{F}}}}^*$$

with pairwise distinct SE failures f_{v_i} and $n \leq |\mathsf{SE}_{\mathcal{F}}|$.

·failure traces

The set of *failure traces* in DFT $\mathcal{F}$ is denoted by $\mathsf{Fails}^{\blacktriangleright}_{\mathsf{SE}_{\mathcal{F}}} \subseteq \mathsf{Events}^{\blacktriangleright}_{\mathcal{F}}$.

For DFTs with *continuous* probability distributions, the failures in failure traces are strictly ordered, because no two SEs can fail at the same time.

Remark 2.2 (No simultaneous failures of SEs) Failures of two SEs cannot happen simultaneously, because the probability of two continuously distributed stochastic events occurring at the same time is zero, cf. Section 2.1.1.2 and [Ste09].

Example 2.3 (Event and failure traces) Consider the DFT in Figure 2.4. A possible ftrace in this DFT is

$$\tau_{\mathsf{f}} = \mathsf{f}_B\ \mathsf{f}_C\ \mathsf{f}_A \in \mathsf{Fails}^{\blacktriangleright}_{\mathsf{SE}}.$$

Trace τ_{f} represents that first SE B fails, then C fails and last, A fails.
A possible event trace for the same DFT is

$$\tau = \upharpoonleft_T\ \upharpoonleft_D\ \upharpoonleft_E\ \upharpoonleft_A\ \upharpoonleft_B\ \upharpoonleft_C\ \mathsf{f}_B\ \mathsf{f}_D\ \oslash_E\ \oslash_T\ \mathsf{f}_C\ \mathsf{f}_E\ \mathsf{f}_T\ \mathsf{f}_A\ \oslash_D.$$

First, all elements are activated. After the failure of SE B, OR-gate D fails. AND-gate E does not fail yet, as its child C is still operational. Thus, no event $\oslash_E$ occurs. TLE T also does not fail yet, i.e. $\oslash_T$. After the failure of C, AND-gate E fails and as a result, TLE T fails as well. The failure of A then has no impact any more.

We often omit the activation events $\uparrow_v$ in the following if they are clear from context.

Valid traces | Note that not all possible event traces correspond to valid behaviour in a DFT. Thus, we define the notion of *valid traces* in the following. These traces represent the behaviours in a DFT which are valid according to a DFT semantics. The definition of valid traces needs several ingredients:

- The behaviour is based on the external failures occurring in the system. We therefore start with a given ftrace τ_f.
- After each SE failure f_v in τ_f, the failure is propagated through the DFT. This propagation results in an event trace $e_1 \dots e_n$. When the propagation is finished, we have obtained a trace $f_v\ e_1 \dots e_n$. Afterwards, the next SE failure in τ_f is considered and propagation starts anew.
- During the propagation, elements in the DFT are evaluated in some evaluation order depending on the used DFT semantics.
- The evaluation of each DFT element leads to a new event, e.g. the failure of the element or no event. The result of the evaluation is uniquely specified by the type of considered DFT element.
- After considering all SE failures in τ_f and performing the necessary propagations, we obtain a valid DFT trace

$$f_v\ e_1 \dots e_n\ f_{v'}\ e'_1 \dots e'_{n'}\ f_{v''} \dots$$

  Note that for a given DFT semantics, multiple evaluation orders might be allowed. Thus, multiple traces might be valid for a given ftrace τ_f.

In the following, we formalize these concepts. We start with the evaluation.

Evaluation of DFT elements | For a given event trace, the next event happening is uniquely determined by the DFT element which is evaluated next. Each type of DFT element gives rise to a specific evaluation.

Definition 2.23 (Evaluation of DFT elements) Let $\mathcal{F}$ be a DFT. The *evaluation* of DFT element $v \in V$ after event trace $\tau \in \mathsf{Events}^{\blacktriangleright}$ is given by the function: evaluation

$$\mathsf{Eval} : \mathsf{Events}^{\blacktriangleright} \times V \longrightarrow \mathsf{Events}.$$

We will define the function Eval for each type of DFT element in Chapter 3.

Observed trace order | For some dynamic gates the order in which failures occur is important. While each trace provides a total order on all occurring events, this can be too strict. We therefore use an ordering which depends on the behaviour which can be observed by each DFT element. Each element can only observe the behaviour of the DFT when it is evaluated. That means for all events occurring between two evaluations of the same element no order can be established and they are considered to be simultaneous.

·events of DFT element

Definition 2.24 (Observed trace order) Let $\mathcal{F}$ be a DFT, $\tau = e_1 \ldots e_n \in \mathsf{Events}^{\blacktriangleright}$ be a trace. For each DFT element $v \in V$, we define the *events of v* as:

$$\mathsf{Events}^v = \begin{cases} \{\mathrm{f}_v, \upharpoonleft_v, \oslash_v\} & \text{if } v \notin \mathsf{SEQ}_{\mathcal{F}} \cup \mathsf{SPARE}_{\mathcal{F}} \\ \{\mathrm{f}_v, \upharpoonleft_v, \oslash_v, \lightning_v\} & \text{if } v \in \mathsf{SEQ}_{\mathcal{F}} \\ \{\mathrm{f}_v, \upharpoonleft_v, \oslash_v\} \cup \left\{\mathrm{cl}^v_{v'} \mid v' \in \sigma(v)\right\} & \text{if } v \in \mathsf{SPARE}_{\mathcal{F}}. \end{cases}$$

observed trace order

Let τ be a trace and v be a DFT element. Let $e, e' \in \tau$ be events with $e, e' \notin \oslash_{\mathcal{F}}$. Trace τ induces an *observed trace order* $\blacktriangleleft^\tau_v$ for v as follows:

$$\begin{aligned} &\forall \tau' e_v \subseteq \tau \text{ with } e_v \in \mathsf{Events}^v \\ &\qquad e \in \tau' \wedge e' \notin \tau' \implies e \blacktriangleleft^\tau_v e' \\ &\qquad e_v \notin \varnothing \wedge e \in \tau' \implies e \blacktriangleleft^\tau_v e_v \\ &\qquad e_v \notin \varnothing \wedge e' \notin \tau' \implies e_v \blacktriangleleft^\tau_v e'. \end{aligned}$$

Each event e_v of v orders the events in those occuring before e_v and events after e_v.

Corollary 2.1 (Observed trace order is a strict partial order) The observed trace order $\blacktriangleleft^\tau_v$ is a *strict partial order*.

·negation

Lemma 2.2 (Negation of observed trace order) The *negation of the observed trace order* is given as follows:

$$\neg\left(e \blacktriangleleft^\tau_v e'\right) \iff e \not\blacktriangleleft^\tau_v e' \vee e \notin \tau \vee e' \notin \tau.$$

Proof. By Definition 2.24 we know:

$$e \blacktriangleleft^\tau_v e' \iff e \blacktriangleleft^\tau_v e' \wedge e \in \tau \wedge e' \in \tau.$$

Correctness of the negation directly follows from negating the right-hand side. ■

Using the notion of observed trace orders allows us to capture different notions of 'simultaneousness' by modifying the order in which elements are evaluated.

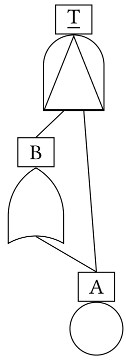


Figure 2.5: Example DFT for observed trace orders

Example 2.4 (Observed trace order) Consider the DFT in Figure 2.5. We consider the two traces $\tau_1 = \mathrm{f}_A\, \mathrm{f}_B\, \mathrm{f}_T$ and $\tau_2 = \mathrm{f}_A \oslash_T \mathrm{f}_B \oslash_T$. The observed trace order for the PAND T induced by the first trace is:

$$\mathrm{f}_A \blacktriangleleft_T^{\tau_1} \mathrm{f}_T \text{ and } \mathrm{f}_B \blacktriangleleft_T^{\tau_1} \mathrm{f}_T.$$

No ordering can be observed between the failures of A and B. From the perspective of PAND T, both children failed simultaneously and thus, the PAND fails as well. For the second trace $\tau_2 = \mathrm{f}_A \oslash_T \mathrm{f}_B \oslash_T$ we get the following observed trace order:

$$\mathrm{f}_A \blacktriangleleft_T^{\tau_2} \mathrm{f}_B.$$

Here, an ordering between T's children can be observed as T is evaluated between the failures of A and B. As a result, A is observed to fail strictly before B. PAND T does not fail, because its children did not fail in the correct order from left to right.

The details of the evaluation function Eval and the observed trace orders are introduced in Chapter 3 when considering each element type in depth. For now, we will only introduce the evaluation function in its general form.

Definition 2.25 (Evaluation of DFT elements) Let $\mathcal{F}$ be a DFT and $\tau \in \mathsf{Events}^{\blacktriangleright}$ be a trace in $\mathcal{F}$. We define the evaluation function for $v \in \mathcal{F}$ as follows:

$$\mathsf{Eval}(\tau, v) = \begin{cases} \mathrm{f}_v & \text{if } \mathrm{f}_v \notin \tau \wedge \mathsf{Fail}(\tau, v) \\ \uparrow_v & \text{else if } \uparrow_v \notin \tau \wedge \mathsf{Act}(\tau, v) \\ \oslash_v & \text{otherwise.} \end{cases}$$

The first case indicates the failure of the gate v if it has not failed before and the failure condition $\mathsf{Fail}(\tau, v)$ holds. The second case represents the activation of v if it is not active yet and condition $\mathsf{Act}(\tau, v)$ holds. We use 'else' here because both conditions could be satisfied at the same time. If none of the cases before holds, no event takes place. The conditions $\mathsf{Fail}(\tau, v)$ and $\mathsf{Act}(\tau, v)$ depend on the type of v and will be defined later in Chapter 3. For now, we only define the failure of static gates.

Definition 2.26 (Failure of static gates) Let $v \in \mathsf{StaticGates}_{\mathcal{F}}$ be a static gate with $n = |\sigma(v)|$ the number of its children. The failure conditions of static gates are:

$$\begin{aligned}
\mathsf{AND}: \quad & \mathsf{Fail}(\tau, v) \iff \bigwedge_{i=1}^{n} \mathrm{f}_{\sigma(v)_i} \in \tau \\
\mathsf{OR}: \quad & \mathsf{Fail}(\tau, v) \iff \bigvee_{i=1}^{n} \mathrm{f}_{\sigma(v)_i} \in \tau \\
\mathsf{VOT}_k: \quad & \mathsf{Fail}(\tau, v) \iff \exists \{i_1, \dots, i_k\} \subseteq \{1, \dots, n\} : \bigwedge_{j=1}^{k} \mathrm{f}_{\sigma(v)_{i_j}} \in \tau.
\end{aligned}$$

The failure condition of an AND-gate is fulfilled iff all children have failed. For an OR-gate, at least one child must have failed and for a VOT_k-gate at least k children must have failed.

Example 2.5 (Evaluation of elements) Consider again the DFT and traces from Example 2.3 on page 32. The next event after the failure of BE B for OR-gate D is given by

$$\mathsf{Eval}(\mathbf{f}_B, D) = \mathrm{f}_D.$$

OR-gate D returns the failure event f_D because its second child B is failed. If we consider the same trace but evaluate AND-gate E instead, no event takes place:

$$\mathsf{Eval}(\mathbf{f}_B, E) = \oslash_E.$$

Evaluation order | The next event is always uniquely determined by the trace considered so far and the DFT element to evaluate next. The order in which DFT elements are evaluated can have severe influences on the induced observed trace order and therefore on the behaviour of the DFT. The evaluation order is specified by the used semantics and can differ quite heavily between different DFT semantics from the literature. A detailed investigation will be provided in Chapter 4. For now it suffices to characterize the evaluation order.

Definition 2.27 (Evaluation order) Let $\mathcal{F}$ be a DFT. Given a DFT semantics *Sem*, the *evaluation order* is defined by the function evaluation order

$$\mathsf{NextEval}^{Sem} : \mathsf{Events}^{\blacktriangleright} \to 2^{V},$$

which returns the possible DFT elements to evaluate next for a given DFT trace.

Given an event trace, an evaluation order defines the next event of the DFT. That means, the evaluation order specified by the semantics defines the exact behaviour of the DFT, i.e. the events occurring.

Definition 2.28 (Next events) Let $\mathcal{F}$ be a DFT. Given a DFT semantics *Sem*, the *next events* after trace $\tau \in \mathsf{Events}^{\blacktriangleright}$ are given by: ·next events

$$\mathsf{NextEvent}^{Sem} : \mathsf{Events}^{\blacktriangleright} \to 2^{\mathsf{Events}}$$
$$\mathsf{NextEvent}^{Sem}(\tau) = \left\{e \mid v \in \mathsf{NextEval}^{Sem}(\tau) \wedge \mathsf{Eval}(\tau, v) = e\right\}.$$

Note that there might be several possible next events, because the evaluation order $\mathsf{NextEval}^{Sem}$ can return multiple elements to evaluate next. It is also possible that there is no possible next event, that is $\mathsf{NextEval}^{Sem}(\tau) = \emptyset$. In that instance, the propagation is finished and the next external failure of an SE can be considered.

Traces in a DFT | We now have all the ingredients to define the behaviour of a DFT for a given DFT semantics. The behaviour of a DFT depends on the external failures of the components, i.e. on the *ftrace* of the SEs. Given an ftrace $\boldsymbol{\tau}_{\mathbf{f}}$ and an evaluation order $\mathsf{NextEval}^{Sem}$—specified by the used semantics *Sem*—we can determine the possible traces in the DFT. We start with an empty trace. We propagate by evaluating the next element given by the evaluation order $\mathsf{NextEval}^{Sem}$. The events computed by $\mathsf{NextEvent}^{Sem}$ are added to the trace as long as $\mathsf{NextEval}^{Sem}$ provides new elements to evaluate. If no elements need to be evaluated any more, the next external failure from ftrace $\boldsymbol{\tau}_{\mathbf{f}}$ is considered and propagation starts anew. The trace is completed, when no propagation can be performed any more and all external failures have been considered.

Definition 2.29 (DFT traces) Let $\mathcal{F}$ be a DFT. We define a recursive helper function which performs the *propagation* in a DFT: ·propagation

$$\mathsf{Propagate}^{Sem}(\tau) = \begin{cases} \bigcup_{e \in \mathsf{NextEvent}^{Sem}(\tau)} \mathsf{Propagate}^{Sem}(\tau\, e) & \text{if } \mathsf{NextEval}^{Sem}(\tau) \neq \emptyset \\ \{\tau\} & \text{otherwise.} \end{cases}$$

We also define the set of event traces which violate a restrictor:

$$\lightning^{\blacktriangleright} = \left\{\tau \in \mathsf{Events}^{\blacktriangleright} \mid \exists v \in \mathsf{SEQ}_{\mathcal{F}} : \lightning_{v} \in \tau\right\}.$$

·valid traces

The set of *valid traces* $\mathsf{Traces}^{Sem}(\boldsymbol{\tau}_\mathbf{f}) \subseteq \mathsf{Events}^{\blacktriangleright} \setminus \lightning^{\blacktriangleright}$ for ftrace $\boldsymbol{\tau}_\mathbf{f} \in \mathsf{Fails}^{\blacktriangleright}_{\mathsf{SE}}$ according to semantics *Sem* is then recursively defined by:

$$\begin{aligned}\mathsf{Traces}^{Sem}(\varepsilon) &= \mathsf{Propagate}^{Sem}(\varepsilon)\\ \mathsf{Traces}^{Sem}(\mathbf{f}_1 \ldots \mathbf{f}_{n-1}\, \mathbf{f}_n) &= \mathsf{Propagate}^{Sem}(\mathsf{Traces}^{Sem}(\mathbf{f}_1 \ldots \mathbf{f}_{n-1})\, \mathbf{f}_n).\end{aligned}$$

·valid traces for semantics

We use Traces^{Sem} to denote the set of *all valid traces* w.r.t. semantics *Sem*:

$$\mathsf{Traces}^{Sem} = \left\{\tau \in \mathsf{Traces}^{Sem}(\boldsymbol{\tau}_\mathbf{f}) \mid \boldsymbol{\tau}_\mathbf{f} \in \mathsf{Fails}^{\blacktriangleright}_{\mathsf{SE}}\right\}.$$

The exclusion of $\lightning^{\blacktriangleright}$ ensures that only traces which satisfy the restrictors are valid. Note also that the initial propagation step is necessary to allow for example initial activation of components.

In the following, we omit the superscript *Sem* and write Traces if we consider all possible semantics. From now on, we use $\tau \in \mathsf{Traces}$ to denote a valid trace τ.

Example 2.6 (DFT traces) We consider again the DFT from Example 2.3 on page 32 for the example semantics *SemEx*. $\mathsf{NextEval}^{SemEx}$ returns element v if one of its children $\sigma(v)_i$ has failed but v has not been evaluated afterwards. Consider the ftrace $\mathbf{f}_B$. To compute $\mathsf{Traces}^{SemEx}(\mathbf{f}_B)$ we start with the empty trace:

$$\mathsf{Traces}^{SemEx}(\varepsilon) = \mathsf{Propagate}^{SemEx}(\varepsilon) = \varepsilon,$$

because $\mathsf{NextEval}^{SemEx}(\varepsilon) = \emptyset$ as no child has failed yet.
Next, we consider the first external failure:

$$\mathsf{Traces}^{SemEx}(\mathbf{f}_B) = \mathsf{Propagate}^{SemEx}(\mathbf{f}_B).$$

As $\mathsf{NextEval}^{SemEx}(\mathbf{f}_B) = \{D, E\}$, we have two possible evaluation orders. We start with D. From Example 2.5 on page 36, we know $\mathsf{Eval}(\mathbf{f}_B, D) = \mathrm{f}_D$ and the resulting trace is $\tau_1 = \mathbf{f}_B\, \mathrm{f}_D$. If we evaluate E instead, we have $\mathsf{Eval}(\mathbf{f}_B, E) = \oslash_E$ and the resulting trace is $\tau_2 = \mathbf{f}_B \oslash_E$. In summary, $\mathsf{NextEvent}^{SemEx}(\mathbf{f}_B) = \{\mathrm{f}_D, \oslash_E\}$ and

$$\mathsf{Propagate}^{SemEx}(\mathbf{f}_B) = \mathsf{Propagate}^{SemEx}(\mathbf{f}_B\, \mathrm{f}_D) \cup \mathsf{Propagate}^{SemEx}(\mathbf{f}_B \oslash_E).$$

We continue with $\tau_1 = \mathbf{f}_B\, \mathrm{f}_D$ and have $\mathsf{NextEval}^{SemEx}(\tau_1) = \{E, T\}$. Possible next events are $\mathsf{NextEvent}^{SemEx}(\tau_1) = \{\oslash_E, \oslash_T\}$ and the corresponding traces are

$$\begin{aligned}\tau_3 &= \tau_1 \oslash_E = \mathbf{f}_B\, \mathrm{f}_D \oslash_E\\ \tau_4 &= \tau_1 \oslash_T = \mathbf{f}_B\, \mathrm{f}_D \oslash_T.\end{aligned}$$

In summary, the propagation step for $\tau_1 = \mathbf{f}_B\, \mathrm{f}_D$ yields

$$\mathsf{Propagate}^{SemEx}(\tau_1) = \mathsf{Propagate}^{SemEx}(\tau_3) \cup \mathsf{Propagate}^{SemEx}(\tau_4).$$

We continue with $\tau_3 = \mathbf{f}_B\, \mathrm{f}_D \oslash_E$. Only one elements needs to be evaluated, because $\mathsf{NextEval}^{SemEx}(\tau_3) = \{T\}$. We obtain trace $\tau_5 = \tau_3 \oslash_T = \mathbf{f}_B\, \mathrm{f}_D \oslash_E \oslash_T$ and get

$$\mathsf{Propagate}^{SemEx}(\tau_3) = \mathsf{Propagate}^{SemEx}(\tau_5).$$

As $\mathsf{NextEval}^{SemEx}(\tau_5) = \emptyset$, we are finished with this trace:

$$\tau_5 = \mathbf{f}_B\, \mathrm{f}_D \oslash_E \oslash_T \in \mathsf{Propagate}^{SemEx}(\tau_5) \subseteq \mathsf{Traces}^{SemEx}(\mathbf{f}_B).$$

In total, we get three possible DFT traces for ftrace $\mathbf{f}_B$:

- $\mathbf{f}_B\, \mathrm{f}_D \oslash_E \oslash_T \in \mathsf{Traces}^{SemEx}(\mathbf{f}_B)$
- $\mathbf{f}_B\, \mathrm{f}_D \oslash_T \oslash_E \in \mathsf{Traces}^{SemEx}(\mathbf{f}_B)$
- $\mathbf{f}_B \oslash_E \mathrm{f}_D \oslash_T \in \mathsf{Traces}^{SemEx}(\mathbf{f}_B)$.

It is important that the propagation only terminates when no changes occur any more. Otherwise, a new external failure would be considered before all failures are fully propagated. Thus, all DFT semantics need to ensure that the evaluation is not stopped prematurely.

Definition 2.30 (Valid evaluation order) Let $\mathcal{F}$ be a DFT and Sem be a DFT semantics. Let $\boldsymbol{\tau}_{\mathbf{f}} = \mathbf{f}_1 \dots \mathbf{f}_n \in \mathsf{Fails}^{\blacktriangleright}_{\mathsf{SE}}$ be any ftrace and let

$$\tau = e_1^0 \dots e_{m_0}^0\, \mathbf{f}_{v_1}\, e_1^1 \dots e_{m_1}^1\, \mathbf{f}_{v_2} \dots \mathbf{f}_{v_n}\, e_1^n \dots e_{m_n}^n \in \mathsf{Traces}^{Sem}(\boldsymbol{\tau}_{\mathbf{f}})$$

be a corresponding DFT trace. For $0 \leq i \leq n$ we define

$$\tau^i = e_1^0 \dots e_{m_0}^0\, \mathbf{f}_{v_1} \dots \mathbf{f}_{v_i}\, e_1^i \dots e_{m_i}^i \subseteq \tau.$$

as the trace until the i-th external failure. For all $e' \in \mathsf{Events}$, we define

$$\tau_{e'}^i = \underbrace{e_1^0 \dots e_{m_0}^0\, \mathbf{f}_{v_1} \dots \mathbf{f}_{v_i}\, e_1^i \dots e_{m_i}^i}_{\tau^i}\, e'\, \mathbf{f}_{v_{i+1}} \dots \mathbf{f}_{v_n}\, e_1^n \dots e_{m_n}^n$$

as the trace where in the i-th propagation one additional event e' is added. We know that each propagation terminated and therefore

$$\forall 0 \leq i \leq n : \mathsf{NextEval}^{Sem}(\tau^i) = \emptyset.$$

valid evaluation order

Semantics *Sem* defines a *valid evaluation order* $\mathsf{NextEval}^{Sem}$ if

$$\forall 0 \leq i \leq n : \forall v \in V : \mathsf{Eval}(\tau^i, v) \in \varnothing,$$

i.e. the propagation can only be prolonged with 'no event', and

$$\forall 0 \leq i \leq n : \forall e' \in \varnothing : \tau^i_{e'} \in \mathsf{Traces}^{Sem}(\boldsymbol{\tau}_{\mathbf{f}}),$$

i.e. prolonging the propagation still yields valid traces.

Example 2.7 (Invalid evaluation order) We consider again the DFT from Example 2.3 on page 32. Let $\boldsymbol{\tau}_{\mathbf{f}} = \mathbf{f}_B\ \mathbf{f}_C$ be the ftrace. We consider the following trace for an example semantics *SemInv*:

$$\mathbf{f}_B\ \varnothing_E\ \mathbf{f}_C\ \mathrm{f}_E\ \varnothing_T \in \mathsf{Traces}^{SemInv}(\boldsymbol{\tau}_{\mathbf{f}}).$$

The evaluation order defined by *SemInv* is invalid, because

$$\mathsf{NextEval}^{SemInv}(\mathbf{f}_B\ \varnothing_E) = \emptyset$$

but

$$\mathsf{Eval}(\mathbf{f}_B\ \varnothing_E, D) = \mathrm{f}_D \notin \varnothing.$$

Remark 2.3 In the remainder, we require that all considered semantics define valid evaluation orders.

2.2.5 State of a DFT

During DFT analysis, we are usually not interested in the complete traces but in the *state* of the complete DFT after a trace, e.g. which elements have failed.

failed elements

Definition 2.31 (Failed elements) Let $\mathcal{F}$ be a DFT and let $\tau \in \mathsf{Traces}$ be a DFT trace in $\mathcal{F}$. The set of *failed elements* after τ is given by

$$\mathsf{Failed}(\tau) = \{v \in \mathcal{F} \mid \mathrm{f}_v \in \tau\}\,.$$

failed

operational

For a trace $\tau \in \mathsf{Traces}$ and element $v \in \mathsf{Failed}(\tau)$ we say 'v is *failed*'. Similarly, for an element $v \in \mathcal{F}$ with $v \notin \mathsf{Failed}(\tau)$ we say 'v is *operational*'.

Monotonicity of failures | In the definition, we use the fact that an element cannot become operational again once it has failed.

Lemma 2.3 (Monotonicity of Failures) Let $\mathcal{F}$ be a DFT. The *monotonicity of failures* is the property that a failed DFT element stays failed forever. More formally, for all DFT elements $v \in \mathcal{F}$ and all traces $\tau \in \mathsf{Traces}$ the following holds: ·monotonicity

$$v \in \mathsf{Failed}(\tau) \implies \forall \tau \subseteq \tau' : v \in \mathsf{Failed}(\tau')$$
$$v \notin \mathsf{Failed}(\tau) \implies \forall \tau'' \subseteq \tau : v \notin \mathsf{Failed}(\tau'').$$

Proof. The proof follows directly from Definition 2.31 and Definition 2.21. ∎

Similar to failed elements, we can also define active elements.

Definition 2.32 (Active elements) Let $\mathcal{F}$ be a DFT and let $\tau \in \mathsf{Traces}$ be a DFT trace in $\mathcal{F}$. The set of *active elements* after τ is given by active elements

$$\mathsf{Active}(\tau) = \{v \in \mathcal{F} \mid \uparrow_v \in \tau\}\,.$$

For an element $v \in \mathsf{Active}(\tau)$ we say 'v is *active*' and for $v \notin \mathsf{Active}(\tau)$ we say 'v is *passive*'. As for failed elements, monotonicity of activation holds as well. active passive

For SPAREs, we define the set of claimed elements.

Definition 2.33 (Claimed elements) Let $\mathcal{F}$ be a DFT and let $\tau \in \mathsf{Traces}$ be a DFT trace in $\mathcal{F}$. The set of *claimed elements* of SPARE s after trace τ is given by claimed elements

$$\mathsf{Claimed}(\tau)(s) : \left\{v \in \mathcal{F} \mid \mathrm{cl}_v^s \in \tau\right\}.$$

For $v \in \mathsf{Claimed}(\tau)(s)$ we say 's has claimed v' and 'v is claimed by s'. Similarly, for an element $v \in \mathcal{F}$ which is not claimed—and also not failed—we say 'v is *available (for claiming)*'. We refer to Definition 3.12 on page 91 for the details.

Remark 2.4 We assume monotonicity for claiming, i.e. elements previously claimed by a SPARE stay claimed—even when the SPARE claims a new child.

Lastly, we consider disabled elements which are not allowed to fail.

Definition 2.34 (Disabled elements) Let $\mathcal{F}$ be a DFT and let $\tau \in \mathsf{Traces}$ be a DFT trace in $\mathcal{F}$. The set of *disabled elements* after τ is defined as follows: disabled elements

$$\mathsf{Disabled}(\tau) = \left\{v \in \mathcal{F} \mid v \notin \mathsf{Failed}(\tau) \wedge \mathsf{Propagate}^{Sem}(\tau\, \mathrm{f}_v) \subseteq ϟ^{\blacktriangleright}\right\}.$$

The failure of disabled elements always yields invalid traces. If a disabled element is a BE, we can therefore prevent occurrences of event ϟ by preventing the disabled BE from failing in the first place. Note that there can be elements which are not disabled but can still yield invalid traces.

2.3 Markov Automata

The base model throughout this thesis is a Markov automaton [EHZ10]. Markov automata combine the non-deterministic and probabilistic behaviour of probabilistic automata [SL95; Sto02] with continuous time as used in continuous-time Markov chains [Nor98].

We base our definitions on [Tim13] in the following.

Markov automaton

Definition 2.35 (Markov automaton (MA)) A *Markov automaton (MA)* is a tuple $\mathcal{M} = (S, s_0, \mathsf{Act}, \hookrightarrow, \dashrightarrow, \mathsf{AP}, L)$, where

- ·states
- ·initial state
- ·actions
- ·probabilistic transitions
- ·Markovian transitions
- ·atomic propositions
- ·state labelling

- S is a finite set of *states*.
- $s_0 \in S$ is the *initial state*.
- Act is a finite set of *actions*.
- $\hookrightarrow \subseteq S \times \mathsf{Act} \times \mathsf{Distr}(S)$ is a *probabilistic transition* relation.
- $\dashrightarrow \subseteq S \times \mathbb{R}_{>0} \times S$ is a *Markovian transition* relation.
- AP is a finite set of *atomic propositions*.
- $L : S \to 2^{\mathsf{AP}}$ is the *state labelling* function.

Actions are also sometimes called choices. For probabilistic transition $(s, \alpha, \mu) \in \hookrightarrow$, we write $s \xhookrightarrow{\alpha} \mu$ and say that from state s with action α we reach a state $s' \in S$ with probability $\mu(s')$. For Markovian transition $(s, \lambda, s') \in \dashrightarrow$, we write $s \overset{\lambda}{\dashrightarrow} s'$ and say that from state s we reach state s' with *rate* λ.

For a state $s \in S$, we denote the *probabilistic transitions* of s by

$$\mathsf{PT}(s) = \{t \in \{s\} \times \mathsf{Act} \times \mathsf{Distr}(S) \mid t \in \hookrightarrow\}.$$

Similarly, we denote the *Markovian transitions* of s by

$$\mathsf{MT}(s) = \{t \in \{s\} \times \mathbb{R}_{>0} \times S \mid t \in \dashrightarrow\}.$$

·probabilistic state
·Markovian state
·hybrid state

We call a state s *probabilistic* if $\mathsf{PT}(s) \neq \emptyset$ and $\mathsf{MT}(s) = \emptyset$. Similarly, a state s is *Markovian* if $\mathsf{MT}(s) \neq \emptyset$ and $\mathsf{PT}(s) = \emptyset$. If both $\mathsf{PT}(s) \neq \emptyset$ and $\mathsf{MT}(s) \neq \emptyset$ hold, then s is a *hybrid state*.

Behaviour of transitions | Both types of transitions in an MA represent different behaviour. In a probabilistic state, first a non-deterministic choice over all outgoing probabilistic transitions is made and we obtain a probability distribution. Afterwards,

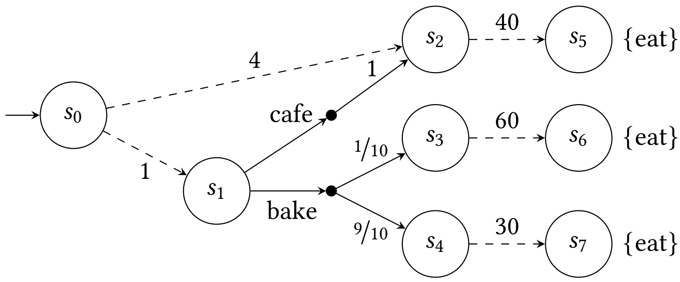


Figure 2.6: Example Markov automaton

the next state is selected according to this discrete probability distribution. In a Markovian state, all outgoing Markovian transitions are competing in a race. Each Markovian transition (s, λ, s') fires after a delay determined by its *rate* λ which is governed by an *exponential distribution*. The transition which fires first, i.e. with the minimal delay, wins the race and determines the successor state. The minimum of the delays is again an exponential distribution and is called the exit rate.

Definition 2.36 (Rates in MA) Let $\mathcal{M} = (S, s_0, \mathrm{Act}, \hookrightarrow, \dashrightarrow, \mathrm{AP}, L)$ be an MA and $s, s' \in S$. The *rate* between states s and s' is given by rate

$$\mathrm{rate}(s, s') = \sum_{(s,\lambda,s') \in \dashrightarrow} \lambda.$$

The *exit rate* (or outgoing rate) of state s is given by ·exit rate

$$\mathrm{rate}(s) = \sum_{s'' \in S} \mathrm{rate}(s, s'').$$

We require $\mathrm{rate}(s) < \infty$ for all $s \in S$.

The time spent in a Markovian state is also called the *sojourn time* and is given by an exponential distribution governed by the exit rate. Thus, the probability to leave state s within t time units is given by the function $1 - e^{-\mathrm{rate}(s) \cdot t}$. After leaving state s, the probability to go to state s' next is given by $\frac{\mathrm{rate}(s,s')}{\mathrm{rate}(s)}$.

Example 2.8 (Markov automaton) We depict an example MA in Figure 2.6. Markovian transitions are depicted with dashed lines. Probabilistic transitions are indicated with a black dot in the middle. The part before the black dot represents the non-deterministic action and the part after the dot represents the probability distribution over the successor states.

The MA models the behaviour of students and faculty for lunch time and the two available options: going to the university cafeteria or the bakery. The initial state s_0 has two outgoing Markovian transitions. The first transition $s_0 \overset{4}{\dashrightarrow} s_2$ models hungry students which go to the university cafeteria (state s_2) with rate 4. The second transition $s_0 \overset{1}{\dashrightarrow} s_1$ models hungry faculty members which emerge with rate 1. A hungry faculty member non-deterministically decides between the two available options represented by the actions 'cafe' and 'bake'. If action 'cafe' is chosen, the faculty member visits the cafeteria (s_2) with probability 1. If action 'bake' is chosen, there is a probability of $1/10$ that the faculty member will be served very fast (s_3). Otherwise, the member will be served with regular speed (s_4). The remaining Markovian transitions model the serving rate in the cafeteria ($s_2 \overset{40}{\dashrightarrow} s_5$) and the fast ($s_3 \overset{60}{\dashrightarrow} s_6$) and slow serving rates ($s_4 \overset{30}{\dashrightarrow} s_7$) in the bakery. In each case, the person can eat afterwards, which is indicated by the atomic property $\{\text{eat}\}$ which is added to each of the states s_5, s_6 and s_7.

maximal progress assumption

Maximal progress assumption | Markov automata conform to the *maximal progress assumption* [Tim13]: probabilistic transitions are immediate and therefore always taken before the (delayed) Markovian transitions. As a result, all Markovian transitions are irrelevant in *hybrid states* and can be omitted. From now on, we therefore assume that Markov automata do not contain hybrid states, i.e.

$$\mathsf{PT}(s) \neq \emptyset \implies \mathsf{MT}(s) = \emptyset \text{ for all } s \in S.$$

·action-deterministic

Action-deterministic | Furthermore, we assume that the MA is *action-deterministic*. That means, for each state s, all outgoing transitions have a unique action, i.e.

$$|\{\alpha \in \mathsf{Act} \mid \exists \mu \in \mathsf{Distr(S)}, (s, \alpha, \mu) \in \mathsf{PT}(s)\}| = 1.$$

2.3.1 Continuous-time Markov chains

An important subclass of Markov automata are continuous-time Markov chains (CTMCs) [Nor98; BHHK03]. CTMCs do not contain non-deterministic choices but only Markovian transitions.

CTMC

Definition 2.37 (Continuous-time Markov chain (CTMC)) A *continuous-time Markov chain (CTMC)* is a tuple $C = (S, s_0, \dashrightarrow, \mathsf{AP}, L)$, where all tuple entries are defined as for Markov automata.

In other words, a CTMC is a Markov automaton with $\mathsf{Act} = \emptyset$ and $\hookrightarrow = \emptyset$.

2.3.2 CSL properties

Schedulers | As we are working with Markov automata, we need to resolve the non-determinism when analysing the model. The non-determinism is resolved by schedulers—often also called policies. We informally introduce the concept of schedulers in the following. For a thorough formalisation, we refer to [Tim13; GHHK+14; Neu10]. A path in the MA is a sequence of states, actions and time spent in each state. A *scheduler* $\mathcal{S}$ determines for each path ending in a state s which action $\mathsf{PT}(s)$ to take next. We denote the set of all schedulers for MA $\mathcal{M}$ by $\mathsf{Schedulers}_{\mathcal{M}}$. Different schedulers can take different actions in a state and therefore lead to different behaviour. For instance in Example 2.8, a scheduler $\mathcal{S}_1$ could always take action *cafe*, i.e. faculty members always go to the cafeteria. Another scheduler $\mathcal{S}_2$ could take action *cafe* with a certain probability and a third scheduler $\mathcal{S}_3$ could take action *bake* if too much time already progressed before.

Continuous stochastic logic | We formalise the *properties*—the measures to compute on our models—in *continuous stochastic logic (CSL)* [BHHK03]. CSL is a branching-time temporal logic based on CTL [EC82; BK08]. We define CSL similar to [BHHK03] and extend it slightly in order to also be applicable to Markov automata. For a formal definition of the properties in the context of Markov automata and an explanation of the corresponding model checking algorithms, we refer to [GHHK+14].

property

continuous stochastic logic (CSL)

Definition 2.38 (Continuous stochastic logic (CSL)) The syntax of *continuous stochastic logic (CSL)* is inductively defined as follows:

·continuous stochastic logic (CSL)
·state formula

- a *state formula* Φ is either
  - an atomic proposition $a \in \mathsf{AP}$,
  - the negation $\neg\Phi_1$ of a state formula Φ_1, or
  - the conjunction $\Phi_1 \wedge \Phi_2$ or disjunction $\Phi_1 \vee \Phi_2$ of state formulas Φ_1, Φ_2.
- a *path formula* φ is of the form $\varphi = \Phi_1 \mathsf{U}^I \Phi_2$ with interval $I \subset \mathbb{R}_{\geq 0}$ and state formulas Φ_1, Φ_2.
- a *probability property* for state s and scheduler $\mathcal{S}$ is of the form $\mathbf{P}_s^{\mathcal{S}}(\varphi)$ with path formula φ.
- an *expected-time property* for state s and scheduler $\mathcal{S}$ is of the form $\mathbf{ET}_s^{\mathcal{S}}(\varphi)$ where the path formula φ has an unbounded interval, i.e. $\varphi = \Phi_1 \mathsf{U}^{[0,\infty)} \Phi_2$.

·path formula

·probability property

·expected-time property

States in an MA are identified in CSL by their *state formula* Φ, a Boolean combination of associated labels.

2

Path formula | A path formula $\varphi = \Phi_1 \mathsf{U}^I \Phi_2$ is satisfied by a path if only states satisfying the state formula Φ_1 are visited until a state satisfying Φ_2 is visited. Moreover, the state satisfying Φ_2 must be visited at a time point $t \in I$ within the given interval I.

·reachability path formula

If $\Phi_1 =$ true for a path formula, then arbitrary states can be visited before satisfying Φ_2. We call this type of path formula a *reachability path formula* and use the notation $\Diamond^I \Phi_2 = \text{true} \, \mathsf{U}^I \Phi_2$. If the interval is of the form $I = [0, t]$, we call the formula a *time-bounded path formula* with time bound t. If the interval is of the form $I = [0, \infty)$, we call the formula an *unbounded path formula* and omit the superscript I.

·time-bounded path formula

·unbounded path formula

Probability property | Probability property $\mathbf{P}_s^{\mathcal{S}}(\varphi)$ describes the probability that the path formula φ is satisfied for scheduler $\mathcal{S}$ starting in state s. We use

$$\mathbf{P}_s^{\max}(\varphi) = \max_{\mathcal{S} \in \mathsf{Schedulers}_{\mathcal{M}}} \mathbf{P}_s^{\mathcal{S}}(\varphi)$$

to denote the maximal probability over all schedulers. Similarly, $\mathbf{P}_s^{\min}(\varphi)$ denotes the minimal probability. We omit the subscript s if we are starting in the initial state s_0, i.e. $\mathbf{P}^{\mathcal{S}}(\varphi) = \mathbf{P}_{s_0}^{\mathcal{S}}(\varphi)$.

Expected-time property | Expected-time properties have the form $\mathbf{ET}_s^{\mathcal{S}}(\Phi_1 \mathsf{U} \Phi_2)$ and describe the expected time when the unbounded path formula $\Phi_1 \mathsf{U} \Phi_2$ is satisfied for scheduler $\mathcal{S}$ starting in state s. As for probability properties, we distinguish maximal and minimal expected times. Note that the expected-time property is undefined if the probability of satisfying the path formula is less than 1, i.e. $\mathbf{P}_s^{\mathcal{S}}(\Phi_1 \mathsf{U} \Phi_2) < 1$.

Expected time properties can be computed on the underlying Markov-decision process (MDP) of a given MA [GHHK⁺14]. As a result, calculating expected-time properties is usually faster than computations on the MA, e.g. for probability properties.

·spurious

Spurious non-determinism | We call the non-determinism in an MA *spurious*, if the computed probabilities are the same for different schedulers, i.e.

$$\mathbf{P}_s^{\min}(\varphi) = \mathbf{P}_s^{\max}(\varphi) \text{ for all path formulas } \varphi.$$

In other words, taking different actions in a state yields the same outcome. For spurious non-determinism, we can the superscript $\mathcal{S}$—or min or max—and write $\mathbf{P}_s(\varphi)$ instead. As CTMCS are deterministic, the minimal and maximal probabilities always coincide.

2.4 Generalised Stochastic Petri Nets

Petri nets [Rei13] are commonly used to model concurrent systems.

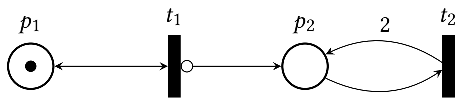


Figure 2.7: Example Petri net

Definition 2.39 (Petri Net) A *Petri net* is a tuple $\mathcal{N} = (P, T, I, O, H, m_0)$, where Petri net

- P is a finite set of *places*. ·places
- T is a finite set of *transitions* with $P \cap T = \emptyset$. ·transitions
- $I, O, H : T \to (P \to \mathbb{N})$ are the input-, output- and inhibition-multiplicities of each transition, respectively.
- $m_0 \in M$ is the initial marking with $M : P \to \mathbb{N}$ the set of markings.

A *marking* $M : P \to \mathbb{N}$ associates each place with a natural number representing the number of *tokens* in this place. ·marking ·token

For a transition $t \in T$ and place $p \in P$ we use the following notation:

- If $I(t)(p) > 0$, there exists an *input arc* from p to t. The input places (or preset) of t are given by ${}^\bullet t = \{p' \in T \mid I(t)(p') > 0\}$. ·input
- If $O(t)(p) > 0$, there exists an *output arc* from t to p. The output places (or postset) of t are given by $t^\bullet := \{p' \in T \mid O(t)(p') > 0\}$. ·output
- If $H(t)(p) > 0$, there exists an *inhibitor arc* from p to t. The inhibition places of t are given by ${}^\circ t := \{p' \in T \mid H(t)(p') > 0\}$. ·inhibitor

Graphical representation | We usually represent Petri nets graphically. Places are depicted with a circle and transitions are depicted with a filled rectangle. Tokens in a place—constituting a marking—are indicated by dots in the corresponding circle. Input arcs are displayed with an arrow from the place to the transition whereas output arcs are displayed with an arrow from the transition to the place. Inhibitor arcs are depicted by an arc from the place to the transition with a small circle as arc-head. Arc-multiplicity greater than one are indicated by a small number near the arc.

Example 2.9 (Petri net) We depict an example Petri net in Figure 2.7. The Petri net consists of two places p_1, p_2 and two transitions t_1, t_2. Transition t_1 has p_1 as input place, both p_1 and p_2 as output places and p_2 as inhibition place. Transition t_2 has place p_2 as input and output place. The input-multiplicity of t_2 is 1 and the output-multiplicity is 2. The initial marking m_0 is given by $m_0(p_1) = 1, m_0(p_2) = 0$.

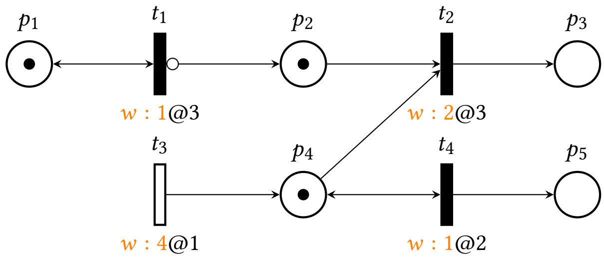


Figure 2.8: Example GSPN

We are using a stochastic extension of Petri nets called Generalised Stochastic Petri Nets (GSPN) [MBCD⁺95] to define the semantics of DFTs in Chapter 3. The following GSPN definition combines the definitions given in [MBCD⁺95] and [EHKZ13].

GSPN

Definition 2.40 (Generalised Stochastic Petri Net (GSPN)) A *Generalised Stochastic Petri Net (GSPN)* is a tuple $\mathcal{G} = (P, T, I, O, H, m_0, W, \Pi, \mathcal{D})$, where

- $\mathcal{N} = (P, T, I, O, H, m_0)$ is a *Petri net*.

·immediate transition
·timed transition

- The transitions $T = T_i \uplus T_t$ are partitioned into *immediate transitions* T_i and *timed transitions* T_t.

·weights

- $W : T \longrightarrow \mathbb{R}_{\geq 0}$ is the *weight function* mapping transitions to weights.

·priorities

- $\Pi : T \longrightarrow \mathbb{N}$ is the *priority function* mapping transitions to a priority level represented by the natural numbers.

·partitioning

- $\mathcal{D} \in 2^{T_i}$ is the *partitioning* of immediate transitions.

Timed transition have priority zero, i.e. $\forall t \in T_t : \Pi(t) = 0$, whereas immediate transitions have a non-zero priority, i.e. $\forall t \in T_i : \Pi(t) > 0$.

Graphical representation | In a graphical depiction of GSPNs, we use filled black bars to depict immediate transitions and white bars with a black border to depict timed transitions. We display transition weights in orange colour with the prefix w and transition priorities with the prefix @. We often omit the weight for a transition t if $W(t) = 1$. The partitioning is not depicted but given separately.

Example 2.10 (GSPN) We depict an example GSPN $\mathcal{G}$ in Figure 2.8. It has three immediate transitions t_1, t_2, t_4 and one timed transition t_3. Transition t_2 for example has weight 2 and priority 3. A possible partitioning is $\mathcal{D} = \{\{t_1\}, \{t_2, t_4\}\}$.

2.4.1 Firing of Transitions

Markings in a GSPN change through the firing of transitions. Only transitions which are in concession and enabled can fire.

Definition 2.41 (Transitions in concession) A transition $t \in T$ has *concession* in a marking $m \in M$ iff ·concession

- $\forall p \in {}^{\bullet}t : m(p) \geq I(t)(p)$ and
- $\forall p \in {}^{\circ}t : m(p) < H(t)(p)$.

A transition has concession if all its input places contain the required amount of tokens and all inhibition places do not exceed the token limit. The set of all transitions which have concession in marking m is denoted by $\mathrm{conc}(m)$.

Definition 2.42 (Enabled transitions) The set of *enabled* transitions in a marking $m \in M$ is given by ·enabled

$$\mathrm{enabled}(m) := \mathrm{conc}(m) \cap \left\{ t \in T \mid \Pi(t) = \max_{t' \in \mathrm{conc}(m)} \Pi(t') \right\}.$$

A transition is enabled if it has concession and no transition of higher priority has concession. As a result, all transitions enabled in a marking have the same priority. Note that if timed and immediate transitions have concession, only the immediate transitions are enabled, because by definition, they have a higher priority. This conforms to the *maximal progress assumption.*

Enabled transition can fire and yield the next marking.

Definition 2.43 (Firing transitions) In a marking $m \in M$, an enabled transition $t \in \mathrm{enabled}(m)$ can *fire*. Firing transition t in m results in a marking m' and is denoted by $m \xrightarrow{t} m'$. The resulting marking m' is given as follows: ·fire

$$\forall p \in P : m'(p) := m(p) + O(t)(p) - I(t)(p).$$

Example 2.11 (Firing transition) We continue with Example 2.10. We first check which transitions have concession in m_0. Transition t_1 has no concession as the inhibitor place p_2 contains a token. Transition t_2 has concession as both input places p_2 and p_4 contain a token. Similarly, transitions t_3 and t_4 also have concession. In total, $\mathrm{conc}(m_0) = \{t_2, t_3, t_4\}$. Of the three transitions, only t_2 is enabled, because it has the highest priority. Thus, $\mathrm{enabled}(m_0) = \{t_2\}$. Firing t_2

consumes a token each of p_2 and p_4, and adds one token to p_3:

$$m_0 \xrightarrow{t_2} m' \text{ with } m'(p_1) = m'(p_3) = 1, m'(p_2) = m'(p_4) = m(p_5) = 0.$$

If multiple transitions are enabled, there is a conflict which transition fires next.

·conflict

Definition 2.44 (Conflict) A marking $m \in M$ contains a *conflict* if multiple transitions are enabled, i.e. $|\mathsf{enabled}(m)| > 1$.

A conflict is resolved by first a non-deterministic choice and secondly a probabilistic choice. For a marking $m \in M$ containing a conflict, we determine the enabled *partitions*:

$$\mathcal{D}_{\mathsf{enabled}}(m) = \{D \in \mathcal{D} \mid D \cap \mathsf{enabled}(m) \neq \emptyset\}\,.$$

We first perform a non-deterministic choice over $\mathcal{D}_{\mathsf{enabled}}(m)$ to determine the partition which is considered next. Let $D \in \mathcal{D}_{\mathsf{enabled}}(m)$ be the chosen partition. Secondly, for all transitions in the partition D, we perform a probabilistic choice w.r.t. *weights*.

·probability of firing

Definition 2.45 (Probability of transition firing) Let $D \in \mathcal{D}_{\mathsf{enabled}}(m)$ be the chosen partition and $t \in D$ be a transition. The *probability of firing* t for marking m and partition D is given by

$$\mathbf{P}_D(t, m) = \frac{W(t)}{\sum_{t' \in D \cap \mathsf{enabled}(m)} W(t')}$$

if $t \in \mathsf{enabled}(m)$ and $\mathbf{P}_D(t, m) = 0$ otherwise.

The probability of a transition is the weight of the transition divided by the total weight of all enabled transitions in the partition.

If only timed transitions are enabled in a marking m, the corresponding *sojourn time* in this marking is exponentially distributed with *exit rate* $\sum_{t \in T_t \cap \mathsf{enabled}(m)} W(t)$.

In summary, conflicts between transitions in different partitions are resolved non-deterministically and conflicts between transitions in the same partition are resolved probabilistically.

Example 2.12 (Conflict) We consider again the GSPN from Example 2.10 but assume that both transitions t_2 and t_4 have priority 3. Thus, both transitions are initially enabled, i.e. $\mathsf{enabled}(m_0) = \{t_2, t_4\}$. We therefore have a conflict in m_0. We resolve this conflict by first considering the partitioning. Both transitions are in the same partition $D = \{t_2, t_4\} \in \mathcal{D}$ and therefore $\mathcal{D}_{\mathsf{enabled}}(m_0) = \{D\}$. As a result, there is no non-deterministic choice as only one partition is enabled.

We continue by calculating the probability of firing for both transitions:

$$\mathbf{P}_D(t_2, m_0) = \frac{2}{2+1} = \frac{2}{3} \qquad\qquad \mathbf{P}_D(t_4, m_0) = \frac{1}{3}.$$

In summary, the conflict between t_2 and t_4 is resolved probabilistically by firing t_2 with probability $2/3$ and firing t_4 with probability $1/3$.

Definition 2.46 (Firing sequence) Let $\mathcal{G}$ be a GSPN. A *firing sequence* starting in marking $m \in M$ is a (finite) sequence of transitions $\sigma = t_1 \dots t_n \in T^*$ such that $m \xrightarrow{t_1} m_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} m_n$ with $m_1, \dots, m_n \in M$. ·firing sequence

A marking m' is *reachable* from marking m if a firing sequence $m \xrightarrow{t_1} \dots \xrightarrow{t_n} m'$ exists. All markings reachable from the initial marking m_0 form the reachability graph. ·reachable

Definition 2.47 (Reachability Graph) Given a GSPN $\mathcal{G}$, the *reachability graph* $\mathrm{RG}(\mathcal{G}) = (\mathrm{RS}(\mathcal{G}), E)$ is a directed graph with ·reachability graph

- $\mathrm{RS}(\mathcal{G})$ the set of reachable markings defined recursively by:
  - $m_0 \in \mathrm{RS}(\mathcal{G})$,
  - if $m \in \mathrm{RS}(\mathcal{G})$ and $\exists t \in T, m' \in M : m \xrightarrow{t} m'$ then $m' \in \mathrm{RS}(\mathcal{G})$.
- $E \subseteq \mathrm{RS}(\mathcal{G}) \times \mathrm{RS}(\mathcal{G})$ the set of edges given by

$$E := \left\{ (m, m') \mid \exists t \in T : m \xrightarrow{t} m' \right\}.$$

Markov automaton for GSPN | A GSPN $\mathcal{G}$ can be translated into an MA $\mathcal{M}_\mathcal{G}$. The idea of the translation as given in [EHKZ13] is as follows: the state space of $\mathcal{M}_\mathcal{G}$ is given by $\mathrm{RS}(\mathcal{G})$, the probabilistic transitions in $\mathcal{M}_\mathcal{G}$ are formed by the immediate transitions of $\mathcal{G}$ and the Markovian transitions are given by the timed transitions.

Properties of GSPN | We consider some properties of GSPNs.

Definition 2.48 (Deadlock) A GSPN $\mathcal{G}$ has a *deadlock* if a marking m is reachable from which no transitions can fire, i.e. $\mathrm{enabled}(m) = \emptyset$. A GSPN is deadlock-free if it does not contain a deadlock. deadlock

Definition 2.49 (Boundedness) Let $k \in \mathbb{N}$ and $\mathcal{G}$ be a GSPN. A place $p \in P$ is k-bounded if for all reachable markings $m \in \mathrm{RS}(\mathcal{G})$: $m(p) \leq k$. $\mathcal{G}$ is k-bounded if all places $p \in P$ are k-bounded. $\mathcal{G}$ is *bounded* if it is k-bounded for some $k \in \mathbb{N}$. ·bounded

While a reachability graph can be infinite in general, it is finite for bounded GSPNs.

Corollary 2.4 (Finite graph) Every bounded GSPN $\mathcal{G}$ has a finite reachability graph $\mathrm{RG}(\mathcal{G})$.

Zeno behaviour

Zeno behaviour | GSPNs (and also Markov automata) can experience *Zeno behaviour*: an infinite sequence of markings (or states) where only a finite amount of time progresses. Zeno behaviour is often considered a modelling error and should be avoided. We characterize Zeno behaviour in GSPNs by the presence of timeless traps [BK02].

·timeless trap

Definition 2.50 (Timeless trap) A GSPN $\mathcal{G}$ has a *timeless trap* if there is a marking $m \in \mathrm{RS}(\mathcal{G})$ from which starts an arbitrarily long firing sequence $m \xrightarrow{t_1} \ldots \xrightarrow{t_n} m_n$ for all $n \in \mathbb{N}$, and all firing sequences $m \xrightarrow{t_1} \ldots \xrightarrow{t_k} m_k$ consist only of immediate transitions: $t_1, \ldots, t_k \in T_i$.

A GSPN with a timeless trap experiences Zeno behaviour as a marking is reachable from which infinitely many immediate transitions fire without any progress of time.

Part II

Semantics

3 Semantics of DFT Gates

Assumptions

In this chapter, we make the following assumptions. The DFT is *well-formed* and satisfies the following additional constraints:

- DFT Constraint 7 (Restriction to exponential distributions)
- DFT Constraint 8 (Independence of modules)
- DFT Constraint 11 (Dependent events of PDEPs are BEs)
- DFT Constraint 13 (Inputs of SEQs are SEs)
- DFT Constraint 14 (No dependencies to inputs of SEQ)

We justify these restrictions throughout this chapter and also discuss approaches to lift them.

Before we can analyse DFTs, we first have to define the precise semantics of DFTs in detail. The semantics of DFTs are usually defined by giving the behaviour of each considered DFT gate, see for example [CSD00; Rai05; BCS10]. However, this does not suffice to fully capture and explain all possible behaviours in a DFT. As already stated by Coppit, Sullivan and Dugan:

> 'A key point is that it is inadequate to specify the meanings of individual modeling constructs in isolation—whether informally or not. What is needed is the definition of the meaning of an arbitrary DFT in which these constructs might interact in subtle ways.' ([CSD00])

That means in addition to specifying the behaviour of each DFT element, we also have to define the interplay between different elements in the whole DFT. This interplay can be specified by the evaluation order. One example is how and in which order failures are propagated from one DFT element to another.

Moreover, there exist different semantics for DFTs in the literature (see Chapter 4 for a detailed discussion). In some details there is no common understanding, what the 'correct' behaviour of a DFT should be.

In this part we aim to tackle these semantic issues. We present an approach to capture the semantics of DFTs based on generalized stochastic Petri nets (GSPNs). We translate a given DFT into a GSPN. As the semantics of GSPNs are precisely defined [EHKZ13], the GSPN then gives the formal semantics of the DFT. As a result, we can give precise semantics to DFTs. Moreover, for different DFT semantics, we provide slightly different translations into GSPNs. Thus, we can compare the differences—and similarities—between DFT semantics by comparing the resulting GSPNs.

We start by giving the semantics of each DFT element in this chapter. For each DFT element we first formalize its behaviour in terms of the evaluation function Eval over *DFT traces* and present possible semantic intricacies. Secondly, we present a GSPN capturing the behaviour of the considered DFT element. We also show the correctness of the resulting GSPN with respect to the DFT traces. We conclude the chapter by showing some properties of the resulting GSPN such as boundedness. The next Chapter 4 then considers the semantics of complete DFTs and discusses the different semantics presented in the literature.

Both chapters are based on the publication 'One Net Fits All - A Unifying Semantics of Dynamic Fault Trees Using GSPNs' [4] and its extended version [5]. The definition of the failure behaviours of DFT elements w.r.t. event traces is inspired by [Jun15]. The approach of translating DFTs to GSPNs was first proposed in [Rai05] and our GSPNs for several DFT elements closely resemble the ones in [Rai05].

3.1 Translation

The translation from DFTs to GSPNs is performed compositionally. The general idea of the translation is visualized in Figure 3.1. We start with the given DFT in Figure 3.1a as input. We independently translate each element of the DFT into a corresponding GSPN using predefined *GSPN templates*. Connections between different GSPN template are handled using *interface places*. The basic scheme of the translation is visualized in Figure 3.1b. For each DFT element, an interface place *Failed* (blue shade) is created. A token in $Failed_v$ represents that DFT element v has failed. Each DFT element is translated using the corresponding GSPN template (cloud shape). Merging all GSPNs yields the complete GSPN for the given DFT. The resulting (simplified) GSPN is depicted in Figure 3.1c. Dashed boxes represent the GSPNs corresponding to different elements.

The presented translation is fully compositional and can be easily adapted and extended. In our translation, we support different DFT semantics by having different GSPN templates and using different *priorities* and *partitionings*. Moreover, new types of DFT elements can easily be added by defining the corresponding GSPN template.

In the following, we first introduce the general idea of the GSPN translation which is based on the concept of GSPN templates. We introduce GSPN templates and explain

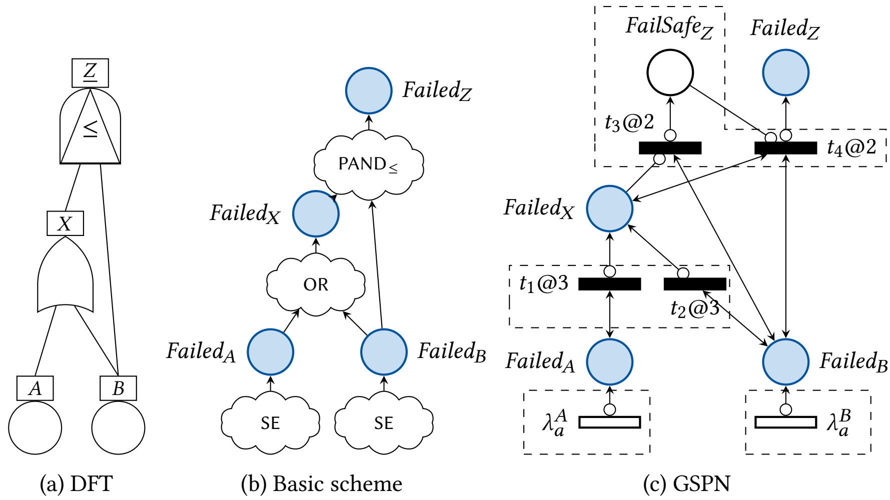


Figure 3.1: Translation idea from DFT to GSPN

how connections between separate GSPNs are handled. Afterwards, we explain the behaviour of each type of DFT element in depth and specify their GSPN templates.

3.2 General Idea

The general idea of the GSPN translation is to construct a correspondence between a given DFT $\mathcal{F}$ and the resulting GSPN $\mathcal{G}_\mathcal{F}$ such that the behaviour of the DFT—given as DFT traces—can be captured by the GSPN—using firing sequences of transitions. The idea is that for each event in the DFT one—or more—corresponding transitions fire in the GSPN. The priorities in the GSPN are used to model the *evaluation order* in the DFT. The transitions which can fire next are determined by the priorities in the GSPN and these priorities are specified by the evaluation order in the DFT. If for example DFT element v is evaluated next in the DFT, the priorities in the GSPN ensure that only transitions corresponding to the GSPN template of v can fire. Thus, the evaluation order is reflected in the corresponding GSPN. Moreover, the current *state* of a DFT, e.g. failed and claimed elements, is captured in the GSPN by its marking.

3.2.1 Interface places

We introduce so-called interface places in the GSPN which represent the *state* of the corresponding DFT. Interface places connect the different DFT elements and are used to pass information from one element to the next.

·interface places

Definition 3.1 (Interface places) Let $\mathcal{F} = (V, \sigma, \mathit{Type}, \mathit{top}, \Theta)$ be a DFT. The set of *interface places* for $\mathcal{F}$ is given by

$$\mathcal{I}_{\mathcal{F}} = \{\mathit{Failed}_v, \mathit{Unavail}_v, \mathit{Active}_v \mid v \in \mathcal{F}\} \cup \{\mathit{Disabled}_v \mid v \in \mathsf{BE}_{\mathcal{F}}\}\,.$$

The interface place Failed_v contains the information about the failure of the DFT element $v \in \mathcal{F}$. A token is in interface place Failed_v iff the corresponding DFT element $v \in \mathcal{F}$ has failed. We also call Failed_v the failure place.

The remaining three types of interface places are needed for specific gates. $\mathit{Unavail}_v$ and Active_v are required for SPARE gates with their claiming and activation mechanism. The place $\mathit{Unavail}_v$ indicates that the corresponding element is not available for claiming, i.e. it is either already failed or it is already claimed by a SPARE gate. The place Active_v indicates whether the element is actively used. An element is actively used if it is either part of the top module or if it belongs to a spare module which is claimed by a SPARE.

The place $\mathit{Disabled}_v$ indicates whether the failure of the corresponding BE is currently prevented by a restrictor, e.g. a SEQ. A *disabled* BE cannot fail.

3.2.2 GSPN templates

The translation from a specific DFT element v to a GSPN is based on GSPN templates $\mathsf{templ}_{\mathsf{type}}(v)$ for each type of DFT element, e.g. AND, OR, etc. Each template uses the predefined interface places to establish the connection to its children. Additionally, a template contains auxiliary places, transitions and arcs.

·priority variables

A template for an element $v \in \mathcal{F}$ also contains a priority variable $\vec{\pi}_v$. The complete set of all *priority variables* is denoted by $\vec{\pi}$. The transition priorities in the resulting GSPN are functions over the priority variables, i.e. $\Pi : T \to \mathbb{N}[\vec{\pi}]$. The priority variables can be instantiated with concrete values from $\mathbb{N}$ to obtain the required transition priorities. Transition priorities specify in which order transitions in the GSPN can fire. We use them to reflect the *evaluation order* in the DFT. Instantiating the priority variables with different values then allows to represent different evaluation orders—i.e. different semantics—without changing the GSPN templates. More details will be given in Section 4.4.

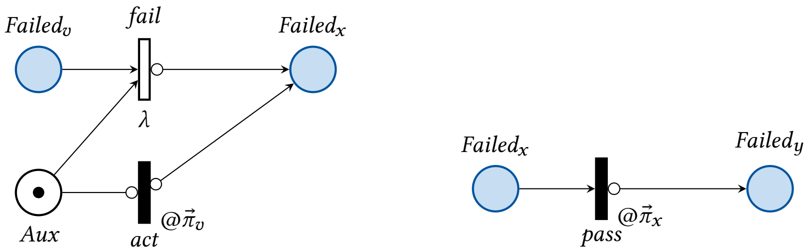
GSPN template

Definition 3.2 (GSPN template) A ($\vec{\pi}$-parameterised) *GSPN template* over interface places $\mathcal{I}$ is a GSPN $\mathcal{T} = (P, T, I, O, H, m_0, W, \Pi, \mathcal{D})$ where

- $\mathcal{I} \subseteq P$ and
- $\Pi : T \to \mathbb{N}[\vec{\pi}]$.

·instantiation

An *instantiation* of GSPN template $\mathcal{T}$ with $c \in \mathbb{N}^n$ results in the GSPN $\mathcal{T}[c]$ =



(a) DFT template $\mathcal{T}_1$

(b) DFT template $\mathcal{T}_2$

Figure 3.2: Example DFT templates

$(P, T, I, O, H, m_0, W, \Pi', \mathcal{D})$ with $\Pi'(t) = \Pi(t)(c)$ for all $t \in T$.

In the following, for ease of presentation, we will often depict the GSPN templates instead of giving their textual description.

Example 3.1 (GSPN template) We depict examples of two GSPN templates in Figure 3.2. The formal definition of the GSPN template in Figure 3.2a is given by $\mathcal{T}_1 = (P, T, I, O, H, m_0, W, \Pi, \mathcal{D})$ over interface places $\mathcal{I} = \{Failed_v, Failed_x\}$ where

- $P = \{Failed_v, Failed_x, Aux\}$,
- $T_t = \{fail\}, T_i = \{act\}$,
- $I(fail)(Failed_v) = 1$, etc.,
- $m_0(Aux) = 1$,
- $W(fail) = 1$,
- $\Pi(act) = \vec{\pi}_v, \Pi(fail) = 0$,
- $\mathcal{D} = T_i$.

In our cases, weights W and partitioning $\mathcal{D}$ are usually only specified when the complete GSPN is constructed.

We still need to describe how to combine several GSPN templates into one GSPN for the complete DFT. As the interface places $\mathcal{I}$ occur in all templates, we merge templates by simply taking the union of places, transitions and arcs.

Definition 3.3 (Merging GSPN templates) For $i = 1, 2$ let GSPN templates $\mathcal{T}_i = (P_i, T_i, I_i, O_i, H_i, m_{0i}, W_i, \Pi_i, \mathcal{D}_i)$ over $\mathcal{I} = P_1 \cap P_2$. The *merge* of $\mathcal{T}_1$ and $\mathcal{T}_2$ results in the template $\text{merge}(\mathcal{T}_1, \mathcal{T}_2) = (P, T, I, O, H, m_0, W, \Pi, \mathcal{D})$ over $\mathcal{I}$ with ·merge

- $P = P_1 \cup P_2$,
- $T = T_1 \uplus T_2$,

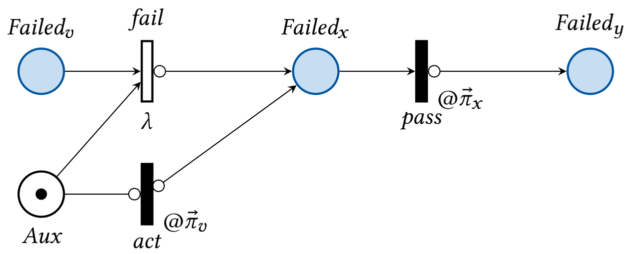


Figure 3.3: Example of merging DFT templates merge$(\mathcal{T}_1, \mathcal{T}_2)$

- $I = I_1 \uplus I_2, \quad O = O_1 \uplus O_2, \quad H = H_1 \uplus H_2,$
- $m_0(p) = \begin{cases} m_{01}(p) + m_{02}(p), & \text{if } p \in \mathcal{I} \\ m_{01}(p), & \text{if } p \in P_1 \setminus \mathcal{I} \\ m_{02}(p), & \text{if } p \in P_2 \setminus \mathcal{I} \end{cases},$
- $W(t) = \begin{cases} W_1(t), & \text{if } t \in T_1 \\ W_2(t), & \text{if } t \in T_2 \end{cases},$
- $\Pi(t) = \begin{cases} \Pi_1(t), & \text{if } t \in T_1 \\ \Pi_2(t), & \text{if } t \in T_2 \end{cases},$
- $\mathcal{D} = \mathcal{D}_1 \uplus \mathcal{D}_2.$

All parts except interface places are disjoint in both input templates. Therefore, the case distinctions for m_0, W and Π are well-defined. Moreover, because of the disjointness, the case distinctions are also commutative and associative. Combined with the fact that (disjoint) union is also commutative and associative, the merge() operation as a whole is commutative and associative as well.

An n-ary merge of templates is obtained by concatenation of the binary merge. For a finite set of templates $\mathcal{T}_1, \dots, \mathcal{T}_n$ over $\mathcal{I}$, the n-ary merge is recursively defined as

$$\text{merge}(\{\mathcal{T}_1, \dots, \mathcal{T}_n\}) = \text{merge}(\mathcal{T}_1, \text{merge}(\mathcal{T}_2, \dots, \mathcal{T}_n)).$$

Example 3.2 (Merging GSPN templates) Consider again the two GSPN templates of Figure 3.2. The result of merging the two templates merge$(\mathcal{T}_1, \mathcal{T}_2)$ is depicted in Figure 3.3. Interface place $Failed_x$ is the only place occurring in both templates and thus, connects both templates in this example.

After defining the general concepts of GSPN templates, we now define the GSPN templates for DFT elements.

Definition 3.4 (GSPN template for DFT element) Let $\mathcal{F}$ be a DFT and $v \in \mathcal{F}$ be an DFT element of type $Type(v) = t$. The *GSPN template for DFT element* v is a ($\vec{\pi}$-parameterised) GSPN template $\mathsf{templ}_t(v)$ over interface places $\mathcal{I}_{\mathcal{F}}$. ·for DFT element

The GSPN template $\mathcal{G}_{\mathcal{F}}$ for a DFT $\mathcal{F}$ mirrors the behaviour of the DFT. A trace τ in $\mathcal{F}$ is mimicked by a *firing sequence* in $\mathcal{G}_{\mathcal{F}}$. Furthermore, the state of a DFT is mimicked by the marking of $\mathcal{G}_{\mathcal{F}}$.

3

Definition 3.5 (Marking after DFT trace) Let $\mathcal{F}$ be a DFT, $\mathcal{G}_{\mathcal{F}}$ be the corresponding GSPN template and $\tau \in \mathsf{Traces}$ be a DFT trace.
The *marking after trace* τ is denoted by $\mathsf{m}(\tau) \in M$ and inductively defined by: marking after trace τ

- For $\tau = \varepsilon$:

$$\mathsf{m}(\varepsilon) = m_0.$$

- For $\tau = \tau' \; e_v$ with $e_v \in \mathsf{Events}^v$:

$$\mathsf{m}(\tau') \xrightarrow{t_1} m_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} m_n = \mathsf{m}(\tau)$$

  where $t_1 \dots t_n$ is a firing sequence in $\mathsf{templ}_{Type(v)}(v)$ and afterwards no transitions of $\mathsf{templ}_{Type(v)}(v)$ are enabled in $\mathsf{m}(\tau)$ any more.

An event e_v for element v in $\mathcal{F}$ is mimicked in $\mathcal{G}_{\mathcal{F}}$ by firing all possible transitions in the GSPN template $\mathsf{templ}_{Type(v)}(v)$ corresponding to v. We know that only one DFT element is evaluated in $\mathcal{F}$ at the same time. Thus, we also need to consider only one GSPN template in $\mathcal{G}_{\mathcal{F}}$. The precise firing transitions depend on the event and the type of DFT element. Details will be given when considering each element type in detail.

The markings in $\mathcal{G}_{\mathcal{F}}$—in particular the interface places—capture the state of the $\mathcal{F}$.

Theorem 3.1 (*Failed*$_v$ interface place) A token is in interface place Failed_v iff the DFT element $v \in \mathcal{F}$ has failed.

$$\forall v \in \mathcal{F} : \forall \tau \in \mathsf{Traces} :$$
$$v \in \mathsf{Failed}(\tau) \iff \mathsf{m}(\tau)(\mathit{Failed}_v) > 0.$$

Proof (sketch). We perform an induction over trace τ.

- $\tau = \varepsilon$: Then $v \notin \mathsf{Failed}(\varepsilon)$ for all $v \in \mathcal{F}$. We ensure by construction that each Failed_v does not contain a token in the initial marking:

$$\mathsf{m}(\varepsilon)(\mathit{Failed}_v) = 0 \text{ for all } v \in \mathcal{F}.$$

- $\tau = \tau' e$: We know
$$v \in \mathsf{Failed}(\tau) \iff \mathsf{f}_v \in \tau.$$
If $\mathsf{f}_v \in \tau'$, then $\mathsf{m}(\tau')(\mathit{Failed}_v) > 0$ holds by induction. As we will later see, tokens in interface places are never removed (cf. Lemma 3.19). It follows
$$\mathsf{m}(\tau)(\mathit{Failed}_v) = \mathsf{m}(\tau')(\mathit{Failed}_v) > 0.$$
For the remainder, we assume $\mathsf{f}_v \notin \tau'$. By induction, we know $\mathsf{m}(\tau')(\mathit{Failed}_v) = 0$. We consider the next event after τ':
$$\mathsf{f}_v \in \mathsf{NextEvent}(\tau') \iff v \in \mathsf{NextEval}(\tau') \wedge \mathsf{Eval}(\tau', v) = \mathsf{f}_v.$$
  - We first consider the case where v is not evaluated, i.e. $v \notin \mathsf{NextEval}(\tau')$. Then $\mathsf{f}_v \notin \mathsf{NextEvent}(\tau')$ and $\mathsf{f}_v \notin \tau'$. By construction of the priorities, not evaluating DFT element v means no transitions in the GSPN template $\mathsf{templ}_{\mathit{Type}(v)}(v)$ can fire in marking $\mathsf{m}(\tau')$. By construction of the GSPN templates, no other GSPN template can add new tokens to Failed_v. Thus,
$$v \notin \mathsf{Failed}(\tau) \implies \mathsf{m}(\tau)(\mathit{Failed}_v) = \mathsf{m}(\tau')(\mathit{Failed}_v) = 0.$$
  - It remains to look at the case where v is evaluated, i.e. $v \in \mathsf{NextEval}(\tau')$. By construction of the priorities, only transitions in $\mathsf{templ}_{\mathit{Type}(v)}(v)$ can fire in marking $\mathsf{m}(\tau')$. It remains to show the following for each DFT element:
$$\begin{aligned} \mathsf{Eval}(\tau', v) = \mathsf{f}_v \iff & \exists t_1, \ldots, t_n \in \mathsf{templ}_{\mathit{Type}(v)}(v) : \\ & \mathsf{m}(\tau') \xrightarrow{t_1} \ldots \xrightarrow{t_n} \mathsf{m}(\tau) \\ & \wedge \mathsf{m}(\tau)(\mathit{Failed}_v) > 0 \end{aligned} \tag{3.1}$$
  where $\mathsf{f}_v \notin \tau'$ and $\mathsf{m}(\tau')(\mathit{Failed}_v) = 0$.

  We prove Equation (3.1) for each GSPN template individually throughout this chapter. ■

The interface place for active elements behaves similar to the failure place.

Theorem 3.2 (Active_v interface place) A token is in interface place Active_v iff the DFT element $v \in \mathcal{F}$ is active. More formally:
$$\begin{gathered} \forall v \in \mathcal{F} : \forall \tau \in \mathsf{Traces} : \\ v \in \mathsf{Active}(\tau) \iff \mathsf{m}(\tau)(\mathit{Active}_v) > 0. \end{gathered}$$

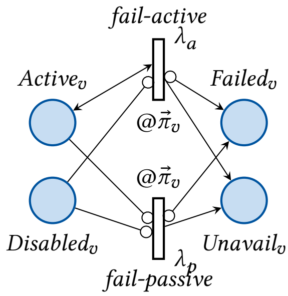


Figure 3.4: GSPN template for SE_{exp}

The proof will be provided later in Section 3.6.4 on page 104 where we consider the GSPN template for activation.

In the rest of the chapter we describe the templates for each element type in detail. In the following, let $\mathcal{F} = (V, \sigma, \mathit{Type}, \mathit{top}, \Theta)$ be a DFT and $\tau \in \mathsf{Traces}$ be a trace in $\mathcal{F}$. For all gates $v \in \mathsf{Gates}_{\mathcal{F}}$, we use $n = |\sigma(v)|$ to denote the number of children of v.

3.3 Basic event

We start by giving the GSPN templates for a basic event. Remember that a BE captures the failure of a concrete atomic component whose failure is not further subdivided. A *stochastic event* SE fails on its own according to some probability distribution. The failure of a SE v is defined by the occurrence of the corresponding event $\mathbf{f}_v \in \mathsf{Events}$. In particular, the evaluation of v will never yield the failure event, because the failures of SEs are already given by the ftrace.

Definition 3.6 (Behaviour of SE) Let $v \in \mathsf{SE}_{\mathcal{F}}$ and $\tau \in \mathsf{Traces}$.

$$\mathsf{Eval}(\tau, v) = \begin{cases} \uparrow_v & \text{if } \uparrow_v \notin \tau \wedge \mathsf{Act}(\tau, v) \\ \oslash_v & \text{otherwise.} \end{cases}$$

The details of the activation relation $\mathsf{Act}(\tau, v)$ are given later in Definition 3.18.

We give the GSPN template $\mathsf{templ}_{\mathsf{SE}_{exp}}(v)$ for an SE v with *exponential* failure distributions $\Theta(v) = \left(\lambda_a, \lambda_p\right)$. The template is depicted in Figure 3.4. It uses the four interface places $Failed_v$, $Unavail_v$, $Active_v$ and $Disabled_v$ and requires no additional auxiliary places. Initially, all places of the template are empty. However, a token is placed in $Active_v$ if the SE_{exp} v becomes active—either due to being part of the top module or due to being used by a SPARE. A token is also placed in $Disabled_v$ if v cannot fail because of a restrictor.

The GSPN template has two *timed transitions fail-active* and *fail-passive*, both with (parameterised) priority $\vec{\pi}_v$. Transition *fail-active* represents the failure of the SE_{exp} v while v is actively used. The active use is ensured through the *input arc* from $Active_v$ to *fail-active*. Moreover, the *inhibitor arc* from $Disabled_v$ to *fail-active* ensures that the SE_{exp} can only fail if it is not disabled by some restrictor. The second *inhibitor arc* coming from $Failed_v$ ensures that a SE_{exp} cannot fail if it has already failed before.

The timed transition *fail-active* fires according to an exponential distribution with rate λ_a. If the transition *fail-active* fires, it places the consumed token back in $Active_v$. It is a common pattern in all the templates that nearly all input arcs also have a corresponding output arc which places the consumed token back into the original place. That way the status of an element—e.g. whether it is active—does not change. Firing *fail-active* also places tokens in $Failed_v$ and $Unavail_v$ indicating that the corresponding SE_{exp} has failed and is therefore also not available for claiming any more. It is a common pattern in our templates that if a token is put in $Failed_v$, a token is also put in $Unavail_v$.

The second timed transition *fail-passive* represents the failure of SE_{exp} v while v is not used. The transition fires according to an exponential distribution with rate λ_p and behaves similar to *fail-active*. The main difference is that *fail-passive* has an *inhibitor* arc from $Active_v$ which enables the transition only if no token is present in $Active_v$, i.e. v is passive.

Proposition 3.3 (At most one timed transition has concession) At most one of the timed transitions *fail-active* and *fail-passive* has concession at the same time.

$$\begin{gathered}\forall m \in M : \\ \mathit{fail\text{-}active} \in \mathrm{conc}(m) \iff \mathit{fail\text{-}passive} \notin \mathrm{conc}(m).\end{gathered}$$

Proof. Let $m \in M$ be an arbitrary marking.

$\Longrightarrow$: Let *fail-active* $\in \mathrm{conc}(m)$. Then it must hold $I(\mathit{fail\text{-}active})(Active_v) = 1 \leq m(Active_v)$. However, $H(\mathit{fail\text{-}passive})(Active_v) = 1 \not> m(Active_v)$ and therefore, *fail-passive* $\notin \mathrm{conc}(m)$.

$\Longleftarrow$: The other direction is similar. ∎

The GSPN template therefore ensures a unique failure mode—*active* or *passive*—of the corresponding SE_{exp} in the DFT.

Correctness of GSPN template | We show that the GSPN template captures the desired behaviour of the SE_{exp}. In particular, firing of a timed transition corresponds to a failure in the corresponding DFT element, and vice versa. We prove Theorem 3.1

for $\mathsf{templ}_{\mathsf{SE}_{exp}}(v)$:

$$\forall v \in \mathsf{SE}_{exp} : \forall \tau \in \mathsf{Traces} :$$
$$v \in \mathsf{Failed}(\tau) \iff \mathsf{m}(\tau)(\mathit{Failed}_v) > 0.$$

Proof. The failure of an SE is already given as part of the ftrace and therefore no evaluation takes place. We therefore cannot use Equation (3.1) to prove the statement. Instead, we perform an induction over the length of the trace $\tau \in \mathsf{Traces}$, very similar to the general proof sketch of Theorem 3.1.

- $\tau = \varepsilon$: Then $v \notin \mathsf{Failed}(\tau) \iff \mathsf{f}_v \notin \tau$. That means, no timed transition of the GSPN template has fired. Further, $\mathsf{m}(\varepsilon) = m_0$ and in particular $m_0(\mathit{Failed}_v) = 0$.
- $\tau = \tau'\ e$: We make a case distinction:
  - If $v \in \mathsf{Failed}(\tau')$, then by induction hypothesis we know:

    $$v \in \mathsf{Failed}(\tau') \implies \mathsf{m}(\tau')(\mathit{Failed}_v) > 0 \implies \mathsf{m}(\tau)(\mathit{Failed}_v) > 0.$$

  - If $v \notin \mathsf{Failed}(\tau')$, then by induction hypothesis it holds $\mathsf{m}(\tau')(\mathit{Failed}_v) = 0$.

    If $v \notin \mathsf{Failed}(\tau)$, then also $e \neq \mathsf{f}_v$ and none of the timed transitions has fired in the GSPN. Therefore, $\mathsf{m}(\tau)(\mathit{Failed}_v) = \mathsf{m}(\tau')(\mathit{Failed}_v) = 0$.

    However, if $v \in \mathsf{Failed}(\tau)$, then $e = \mathsf{f}_v$ and DFT element v has just failed in the last step. That means one of the timed transitions *fail-active* or *fail-passive* has fired. At least one of the transitions is enabled, because Failed_v does not contain a token yet. The failure is also not disabled, because by Definition 2.34 we have $v \in \mathsf{Failed}(\tau) \implies v \notin \mathsf{Disabled}(\tau)$. As later shown in Theorem 3.15 on page 119 it follows $v \notin \mathsf{Disabled}(\tau) \implies \mathsf{m}(\tau)(\mathit{Disabled}_v) = 0$. If Active_v contains a token, *fail-active* is enabled, otherwise *fail-passive* is enabled. Firing the transition puts a token in Failed_v and thus, $\mathsf{m}(\tau)(\mathit{Failed}_v) = 1$. ∎

We conclude this section by presenting simplified templates for special cases of BEs.

Cold $\mathbf{SE}_{exp}$ | If the SE_{exp} cannot fail in standby/passive mode, we call it a *cold* SE_{exp}. The corresponding GSPN template is given in Figure 3.5. The template has no *fail-passive* transition any more as only failures in active mode are possible. Note that in principle, an ftrace could contain the failure of a cold SE_{exp} in passive mode—even though the cold SE_{exp} has a zero probability of failure. However, this specific ftrace would have a probability of zero and can therefore never occur. ·cold

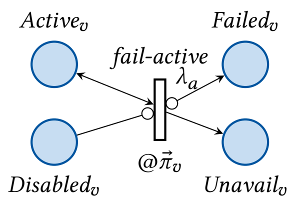
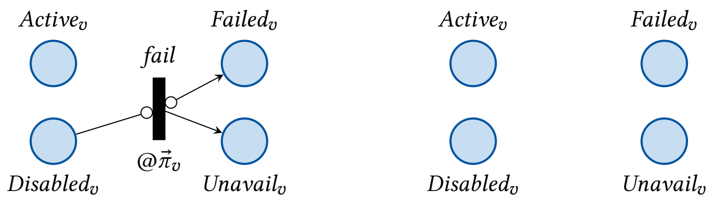


Figure 3.5: GSPN template for cold SE_{exp}



(a) Constant failed $\mathsf{CONST}_\top$

(b) Constant fail-safe $\mathsf{CONST}_\bot$

Figure 3.6: GSPN templates for constant BE

Constant BE | *Constant failed* and *constant fail-safe* BE experience the following failure behaviour.

Definition 3.7 (Failure behaviour of constant BE)

$$\begin{aligned} \mathsf{CONST}_\top : &\quad \mathsf{Fail}(\tau, v) \\ \mathsf{CONST}_\bot : &\quad \neg\mathsf{Fail}(\tau, v). \end{aligned}$$

Note that $\mathsf{CONST}_\top$ is not failed initially but fails only if it is evaluated for the first time. This allows more flexibility in failure propagation by respecting the evaluation order. For example, in case of multiple $\mathsf{CONST}_\top$, we ensure a strict ordering of the failures instead of having to cope with simultaneous failures of BEs (cf. [JGKS16]).

The template $\mathsf{templ}_{\mathsf{CONST}_\top}(v)$ for the constant failed BE is given in Figure 3.6a. It consists of one *immediate transition fail*. The transition is initially enabled and its firing places tokens in *Failed*$_v$ and *Unavail*$_v$ indicating the immediate failure of v.

The template $\mathsf{templ}_{\mathsf{CONST}_\bot}(v)$ for the constant fail-safe BE is given in Figure 3.6b. The template only contains the interface places but no transitions as the element cannot fail on its own. The interface places always must be present in a template. Additionally, even a $\mathsf{CONST}_\bot$ might become failed because of an outside trigger from an FDEP. Note that a $\mathsf{CONST}_\bot$ is syntactic sugar and can for example be modelled with PANDs [Jun15], cf. Figure 4.1 on page 134.

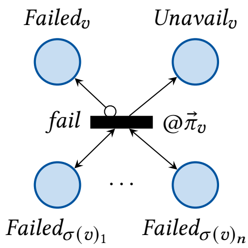


Figure 3.7: GSPN template for AND

Other probability distributions | We restrict ourselves to exponential distributions here. It is certainly possible to lift this restriction though. For example, *Erlang* distributions can be modelled using multiple timed transitions in sequence [MBCD+95]. In general, every *phase-type* distribution can be modelled by a GSPN [BCB89]. Discrete probability distributions such as *Bernoulli* distributions can be modelled by setting specific weights for immediate transitions. In general, more probability distributions can be supported in our framework by defining the corresponding GSPN template.

3.4 Static Gates

The failure behaviour of static gates [VGRH81] follows their corresponding Boolean operators. An AND-gate fails if all its inputs fail, an OR-gate fails if at least one of its inputs fails and a VOT_k-gate fails if at least k of its inputs fail. We defined the precise failure behaviour of the static gates in Definition 2.26 on page 36 already.

Note that in the following templates the interface place $Active_v$ is not depicted. The GSPN extensions for activation of gates are introduced later in Section 3.6.4 on page 104.

3.4.1  AND gate

The template $\mathsf{templ}_{\mathsf{AND}}(v)$ for an AND with n children is given in Figure 3.7. The GSPN template depicts the two interface places $Failed_v$ and $Unavail_v$ of the AND-gate v. Additionally, the interface places $Failed_{\sigma(v)_1}, \ldots, Failed_{\sigma(v)_n}$ containing the failure status of the children are present. The GSPN template contains one immediate transition *fail*. The immediate transition has the usual construction with output arcs going to $Failed_v$ and $Unavail_v$, and an inhibitor arc to $Failed_v$. The inhibitor arc prevents the transition from firing multiple times if the tokens are already indicating a failure. Transition *fail* has input arcs from and output arcs to all children. That way the transition is only in concession iff all children $\sigma(v)_i$ have a token in their $\mathit{Failed}_{\sigma(v)_i}$ place. The output arcs ensure that the tokens consumed by firing *fail* are also placed back in the same place. That way DFT elements which are input to multiple gates can propagate their failure to all of the gates one after the other.

Correctness of GSPN template | We prove Theorem 3.1 for $\mathsf{templ}_{\mathsf{AND}}(v)$:

$$\forall v \in \mathsf{AND} : \forall \tau \in \mathsf{Traces} :$$
$$v \in \mathsf{Failed}(\tau) \iff \mathsf{m}(\tau)(\mathit{Failed}_v) > 0.$$

Proof. It suffices to show Equation (3.1) on page 62:

$$\mathsf{Eval}(\tau', v) = \mathsf{f}_v \iff \exists t_1, \ldots, t_n \in \mathsf{templ}_{\mathit{Type}(v)}(v) : \mathsf{m}(\tau') \xrightarrow{t_1} \ldots \xrightarrow{t_n} \mathsf{m}(\tau)$$
$$\wedge\, \mathsf{m}(\tau)(\mathit{Failed}_v) > 0.$$

We prove both cases.

- If $\mathsf{Eval}(\tau', v) = \mathsf{f}_v$, then by definition it holds

$$\mathsf{Fail}(\tau', v) \implies \bigwedge_{i=1}^{n} \mathsf{f}_{\sigma(v)_i} \in \tau'.$$

  By induction we know

$$\mathsf{f}_{\sigma(v)_i} \in \tau' \implies \mathsf{m}(\tau')(\mathit{Failed}_{\sigma(v)_i}) > 0.$$

  In marking $\mathsf{m}(\tau')$, immediate transition *fail* is in concession, because each input place $\mathit{Failed}_{\sigma(v)_i}$ contains a token. Moreover, Failed_v cannot contain a token yet, because $\mathsf{m}(\tau')(\mathit{Failed}_v) = 0$. The firing sequence in the GSPN template is

$$\mathsf{m}(\tau') \xrightarrow{\mathit{fail}} \mathsf{m}(\tau).$$

  Firing transition *fail* places a token in Failed_v and therefore $\mathsf{m}(\tau)(\mathit{Failed}_v) > 0$.

- If $\mathsf{Eval}(\tau', v) \neq \mathsf{f}_v$, then $\neg\mathsf{Fail}(\tau', v)$, because by assumption $\mathsf{f}_v \notin \tau'$. Then there is a $1 \leq j \leq n$ such that $\mathsf{f}_{\sigma(v)_j} \notin \tau'$. By induction hypothesis, $\mathsf{m}(\tau')(\mathit{Failed}_{\sigma(v)_j}) = 0$ and therefore transition *fail* is not enabled. Due to the *monotonicity* of failures, *fail* was never enabled and no token was ever placed in Failed_v. Therefore, $\mathsf{m}(\tau)(\mathit{Failed}_v) = 0$. ∎

The proof uses the fact that the AND-gate is static, i.e. only the failure of an input but

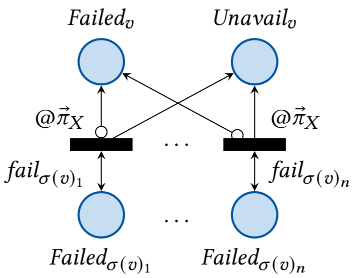


Figure 3.8: GSPN template for OR

not the order of failures is relevant.

3.4.2 OR gate

The template $\mathsf{templ}_{\mathsf{OR}}(v)$ for an OR with n children is depicted in Figure 3.8. The main difference to the AND-gate is that each input of the OR-gate has its own immediate transition $fail_{\sigma(v)_i}$. As soon as one of the inputs $\sigma(v)_i$ is failed, the corresponding transition $fail_{\sigma(v)_i}$ is in concession and can place a token in $Failed_v$. If another input fails as well, the inhibitor arc prevents a second transition from firing and only one token will ever be present in the failure place.

Correctness of GSPN template | We prove Theorem 3.1 for $\mathsf{templ}_{\mathsf{OR}}(v)$:

$$\forall v \in \mathsf{OR} : \forall \tau \in \mathsf{Traces} :$$
$$v \in \mathsf{Failed}(\tau) \iff \mathsf{m}(\tau)(Failed_v) > 0.$$

> *Proof (sketch).* The proof is similar to the proof for the AND-gate. ∎

3.4.3 Voting gate (VOT)

The voting gate VOT is a generalization of AND and OR. Both AND and OR can be expressed using VOT-gates.

> **Lemma 3.4** An AND-gate can be expressed by a VOT_n-gate and an OR-gate can be expressed by a VOT_1-gate.

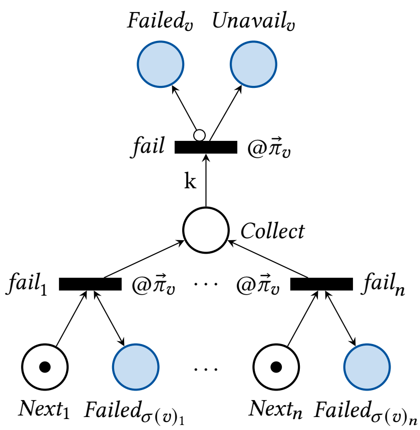


Figure 3.9: GSPN template for VOT_k

Proof. The correctness follows from Definition 2.26. For VOT_n we get:

$$\mathsf{Fail}(\tau, v) = \exists \{i_1, \ldots, i_n\} \subseteq \{1, \ldots, n\} : \bigwedge_{j=1}^{n} \mathrm{f}_{\sigma(v)_{i_j}} \in \tau = \bigwedge_{j=1}^{n} \mathrm{f}_{\sigma(v)_{i_j}}.$$

For VOT_1 we get:

$$\mathsf{Fail}(\tau, v) = \exists \{i_1\} \subseteq \{1, \ldots, n\} : \bigwedge_{j=1}^{1} \mathrm{f}_{\sigma(v)_{i_j}} \in \tau = \bigvee_{i_1 \in \{1,\ldots,n\}} \mathrm{f}_{\sigma(v)_{i_j}} \in \tau. \qquad \blacksquare$$

The template $\mathsf{templ}_{\mathsf{VOT}_k}(v)$ for a VOT_k-gate with n children and threshold k is depicted in Figure 3.9. The template contains additional places $Next_1, \ldots, Next_n$ and *Collect*. These auxiliary places are not interface places and only used within a specific voting gate. Place *Collect* is one of the few places within the complete GSPN framework which can contain more than one token. It collects tokens for every failed child. If the number of tokens—the number of failed children—is at least k, the immediate transition *fail* can fire. The threshold k is ensured by the input multiplicity k of *fail*.

A token for the failure of child $\sigma(v)_i$ is placed in *Collect* through transition $fail_i$. The transition consumes the token from the auxiliary place $Next_i$ and therefore ensures that only one token can be placed in *Collect* for each failed child. Note that we always put the token back into $Failed_{\sigma(v)_i}$. Without place $Next_i$, transition *faili* could therefore fire arbitrary many times.

Correctness of GSPN template | We prove Theorem 3.1 for $\mathsf{templ}_{\mathsf{VOT}_k}(v)$:

$$\forall v \in \mathsf{VOT}_k : \forall \tau \in \mathsf{Traces} :$$
$$v \in \mathsf{Failed}(\tau) \iff \mathsf{m}(\tau)(\mathit{Failed}_v) > 0.$$

Proof. We prove again both cases of Equation (3.1) on page 62:

$$\mathsf{Eval}(\tau', v) = \mathsf{f}_v \iff \exists t_1, \dots, t_n \in \mathsf{templ}_{\mathit{Type}(v)}(v) : \mathsf{m}(\tau') \xrightarrow{t_1} \dots \xrightarrow{t_n} \mathsf{m}(\tau)$$
$$\land\, \mathsf{m}(\tau)(\mathit{Failed}_v) > 0.$$

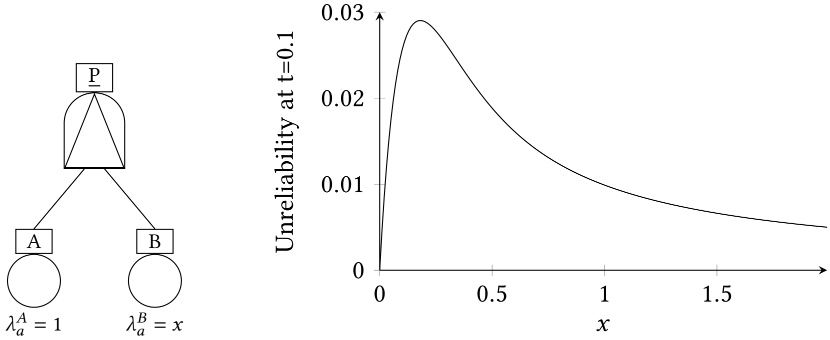
- If $\mathsf{Eval}(\tau', v) = \mathsf{f}_v$, then there exist k children which have failed. W.l.o.g. we assume the first k children have failed, i.e. $\bigwedge_{i=1}^{k} \mathsf{f}_{\sigma(v)_i} \in \tau'$. By induction hypothesis, $\mathsf{m}(\tau')(\mathit{Failed}_{\sigma(v)_i}) > 0$ for all $1 \leq i \leq k$. We consider each of the first k fail_i transitions. If Next_i still contains a token, the transition fail_i can fire—because $\mathit{Failed}_{\sigma(v)_i}$ also contains at least one token. Firing fail_i puts an additional token in *Collect*. If Next_i contains no token, the transition fail_i must have fired before and *Collect* already contains the corresponding token. The argument holds for the first k children and therefore *Collect* must contain at least k tokens. Thus, transition *fail* can fire and as a result $\mathsf{m}(\tau)(\mathit{Failed}_v) > 0$.

- If $\mathsf{Eval}(\tau', v) \neq \mathsf{f}_v$, then there exist at least $n{-}k{+}1$ children which are not failed yet. W.l.o.g. we assume the first $k{-}1$ children are failed and the children $\sigma(v)_k, \dots, \sigma(v)_n$ are operational. Following the argumentation from before, we have at least $k{-}1$ tokens in *Collect* coming from the first $k{-}1$ failed children. None of the other transitions $\mathit{fail}_k, \dots, \mathit{fail}_n$ are in concession, because $\mathsf{m}(\tau')(\mathit{Failed}_{\sigma(v)_i}) = 0$ for all $k \leq i \leq n$. Therefore $\mathsf{m}(\tau')(\mathit{Collect}) \leq k - 1$ and *fail* is not enabled. As a result $\mathsf{m}(\tau)(\mathit{Failed}_v) = \mathsf{m}(\tau')(\mathit{Failed}_v) = 0$. ∎

The gate types considered so far—AND, OR, VOT—are *static* gates where only the fact is relevant whether an element has failed or not. A fault tree consisting only of static gates and BEs is a *static fault tree (SFT)*. The remainder of the chapter considers dynamic gates where also the order of failures is relevant. SFT

3.5 Priority Gates

We start with dynamic extensions of AND- and OR-gates where the failures have to occur according to a certain priority order. The priority AND gate (PAND) [FAR76] fails only if the failures of the children occur in order from left to right, i.e. child $\sigma(v)_1$ fails before $\sigma(v)_2$, and so forth. Similarly, the priority OR gate (POR) [WP09] fails only if the first child fails first. If one of the children of a priority gate fails out of order, the



(a) DFT (b) Reliability w.r.t. failure rate of B

Figure 3.10: Non-coherence for PAND

gate becomes *fail-safe* and can never fail any more.

coherence

Coherence | One important aspect of priority gates is that they are in general not coherent [CADC18]. *Coherence* requires two properties to hold [CADC18; TB17, Chapter 6.2]:

·relevance

·monotonicity

1. *Relevance*: All components contribute to the system reliability.
2. *Monotonicity*: Improving any component leads to an improved system reliability.

In SFTs, Boolean constructs such as NOT or XOR cause non-coherence [TB17]. Priority gates can also be non-coherent, because monotonicity is not always satisfied. Decreasing the reliability of the right-most child makes it more likely that the failure order of the priority gate is violated and the gate becomes fail-safe. As a result the system becomes more reliable.

Example 3.3 (Coherence of PAND) We depict one example of non-coherence in Figure 3.10. Consider the simple DFT consisting of one PAND with two children in Figure 3.10a. We change the failure rate x of the right child B and compute the resulting unreliability of PAND P. The results are visualized in Figure 3.10b. The x-axis depicts the failure rate $\lambda_a^B = x$ of B and the y-axis gives the unreliability of P at time $t = 0.1$. The failure rate of A is fixed to $\lambda_a^A = 1$. The plot shows that for small failure rates, the system behaves coherent: increasing the failure rate of B also increases the unreliability of the system. However, at approximately $x \approx 0.2$ the behaviour changes and is not monotonic any more. Increasing the failure rate of B now decreases the unreliability of the system. The reason is that it becomes more likely that B fails before A and therefore the PAND becomes fail-safe.

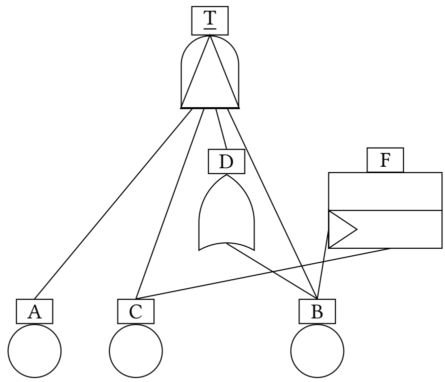


Figure 3.11: PAND depicting ordering intricacies

When using priority gates, it is thus important to be aware of coherence and carefully reflect whether non-coherent behaviour is desired. Otherwise, the DFT needs to be adapted accordingly.

Before defining the precise failure behaviour of priority gates, two questions about special cases in the failure ordering need attention.

Simultaneous failures | The first question is how simultaneous failures of children are handled. As an example, consider the DFT given in Figure 3.11 and the ftrace $\boldsymbol{\tau}_{\mathsf{f}} = \mathsf{f}_A\ \mathsf{f}_C\ \mathsf{f}_B$. We consider the trace $\tau_1 = \mathsf{f}_A \oslash_T \mathsf{f}_C \oslash_T \mathsf{f}_B\ \mathsf{f}_D \in \mathsf{Traces}(\boldsymbol{\tau}_{\mathsf{f}})$. Here, BE A fails first and as it is the first child, the ordering requirement of the PAND T is respected. Next, BE C fails—also in the correct order. Last, BE B fails which leads to the immediate failure of the OR-gate D. The observed trace order for PAND T is as follows:

$$\mathsf{f}_A \blacktriangleleft_T^{\tau_1} \mathsf{f}_C \blacktriangleleft_T^{\tau_1} \mathsf{f}_B \text{ and } \mathsf{f}_C \blacktriangleleft_T^{\tau_1} \mathsf{f}_D.$$

From the perspective of PAND T, both B and D have failed simultaneously, because no ordering could be observed. The question now is whether this simultaneous failure satisfies the order of failures from left to right, i.e. should PAND T be failed or fail-safe?

This question was first stated in [CSD00], where it came up in a real-life model created by an engineer from Lockheed-Martin.

We solve this question by presenting two different types of priority gates: inclusive and exclusive priority gates. The *inclusive* priority gate allows simultaneous failures and considers them to be in order. In contrast, the *exclusive* priority gate only fails if failures occur strictly from left to right with no simultaneous failures. We depict inclusive priority gates with the symbol $\leq$ and exclusive priority gates with $<$.

·inclusive
·exclusive

Causal ordering of failures | The second question is whether the ordering of the failures of the children is a temporal ordering or whether the ordering must also respect causality. To better understand this question, we again consider the example DFT in Figure 3.11. We consider the ftrace $\boldsymbol{\tau}'_{\mathrm{f}} = \mathbf{f}_A\, \mathbf{f}_B$. BE A fails first and then B fails, rendering D to fail. Additionally, the failure of B triggers the failure of C through the functional dependency F. From a causal point of view, B failed strictly before C, because without the failure of B, the BE C would not have failed. However, from a temporal point of view, no time has passed between the failures of B and C, because C fails immediately after B. Thus, their failure can also be considered simultaneous. The question is again whether the failures of B and C respect the failure order, i.e. should PAND T be failed or fail-safe?

We solve this question by using the evaluation order—and the inferred observed trace order—to decide which failure order should be respected. In our example, several traces with different observed trace orders are possible:

$$
\begin{array}{ll}
\tau_2 = \mathbf{f}_A \oslash_T \mathbf{f}_B\, \mathrm{f}_D\, \mathrm{f}_C \in \mathsf{Traces}(\boldsymbol{\tau}'_{\mathrm{f}}) & \mathbf{f}_A \blacktriangleleft^{\tau_2}_T \mathbf{f}_B, \mathbf{f}_A \blacktriangleleft^{\tau_2}_T \mathrm{f}_D, \mathbf{f}_A \blacktriangleleft^{\tau_2}_T \mathrm{f}_C \\
\tau_3 = \mathbf{f}_A \oslash_T \mathbf{f}_B\, \mathrm{f}_D \oslash_T \mathrm{f}_C \in \mathsf{Traces}(\boldsymbol{\tau}'_{\mathrm{f}}) & \mathbf{f}_A \blacktriangleleft^{\tau_3}_T \mathbf{f}_B, \mathbf{f}_A \blacktriangleleft^{\tau_3}_T \mathrm{f}_D, \mathbf{f}_B \blacktriangleleft^{\tau_3}_T \mathrm{f}_C, \mathrm{f}_D \blacktriangleleft^{\tau_3}_T \mathrm{f}_C \\
\tau_4 = \mathbf{f}_A \oslash_T \mathbf{f}_B \oslash_T \mathrm{f}_D \oslash_T \mathrm{f}_C \in \mathsf{Traces}(\boldsymbol{\tau}'_{\mathrm{f}}) & \mathbf{f}_A \blacktriangleleft^{\tau_4}_T \mathbf{f}_B \blacktriangleleft^{\tau_4}_T \mathrm{f}_D \blacktriangleleft^{\tau_4}_T \mathrm{f}_C \\
\tau_5 = \mathbf{f}_A \oslash_T \mathbf{f}_B \oslash_T \mathrm{f}_C \oslash_T \mathrm{f}_D \in \mathsf{Traces}(\boldsymbol{\tau}'_{\mathrm{f}}) & \mathbf{f}_A \blacktriangleleft^{\tau_5}_T \mathbf{f}_B \blacktriangleleft^{\tau_5}_T \mathrm{f}_C \blacktriangleleft^{\tau_5}_T \mathrm{f}_D
\end{array}
$$

Trace τ_2 encodes only the temporal ordering. In this trace, B, C and D have failed simultaneously and therefore PAND T fails. In contrast, trace τ_3 encodes both the temporal and causal ordering. In τ_3, BE C fails strictly after B and D and therefore PAND T becomes fail-safe. The other two traces τ_4 and τ_5 yield a total order of the observed events and both lead to a fail-safe PAND.

In summary, all possible behaviours can be modelled by providing the right evaluation order. We consider the handling of causal orderings in detail in Section 4.4.2 on page 163 and present valid evaluation orders for different semantics from the literature.

3.5.1 Priority AND (PAND)

The priority AND-gate (PAND) fails if all children fail in order from left to right.

3.5.1.1 Inclusive PAND

·inclusive We start by introducing $\mathrm{PAND}_{\leq}$, the *inclusive* PAND-gate, where simultaneously occurring failures of children satisfy the priority order as well. In other words, the failure order is respected as long as no child fails strictly before its left sibling.

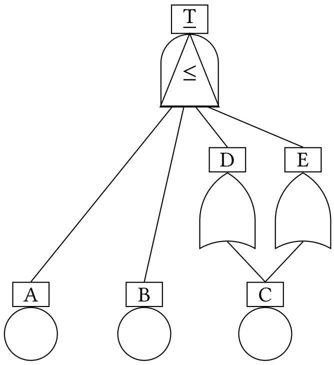


Figure 3.12: Example DFT with inclusive $\mathsf{PAND}_{\leq}$

Definition 3.8 (Failure of $\mathsf{PAND}_{\leq}$) Let $v \in \mathsf{PAND}_{\leq}$ and $\tau \in \mathsf{Traces}$.

$$\mathsf{Fail}(\tau, v) \iff \bigwedge_{i=1}^{n} \mathrm{f}_{\sigma(v)_i} \in \tau \wedge \bigwedge_{i=2}^{n} \mathrm{f}_{\sigma(v)_i} \not\blacktriangleleft_v^{\tau} \mathrm{f}_{\sigma(v)_{i-1}}.$$

The failure condition of $\mathsf{PAND}_{\leq}$ can be separated into two parts: the first part is the standard failure condition of the AND-gate stating that all children have to fail. The second part defines the ordered failure of the children. The i-th child $\sigma(v)_i$ cannot fail strictly before the left sibling $\sigma(v)_{i-1}$ has failed.

The observed trace order $\blacktriangleleft_v^{\tau}$ is induced by the previous evaluations of PAND v. The failure behaviour of PAND depends not only on the current state of the elements but also on the previous states whenever v was evaluated. That means we also need to consider the failure history which yields the dynamic behaviour of the PAND-gate.

Example 3.4 (PAND) Consider the DFT with a $\mathsf{PAND}_{\leq}$-gate in Figure 3.12. We consider the ftrace $\boldsymbol{\tau}_{\mathbf{f}} = \mathbf{f}_A\ \mathbf{f}_B\ \mathbf{f}_C$. One possible corresponding trace is

$$\tau' = \mathbf{f}_A \oslash_T \mathbf{f}_B \oslash_T \mathbf{f}_C\ \mathrm{f}_D\ \mathrm{f}_E\ \mathrm{f}_T \in \mathsf{Traces}(\boldsymbol{\tau}_{\mathbf{f}}).$$

In trace τ', A fails first and B fails afterwards. Then C fails and immediately the OR-gates D and E fail as well. Therefore, C, D and E have failed simultaneously. As the failure order of the $\mathsf{PAND}_{\leq}$ is respected, T fails as well and $T \in \mathsf{Failed}(\tau')$.

The template $\mathsf{templ}_{\mathsf{PAND}_{\leq}}(v)$ for an inclusive PAND with n children is given in Figure 3.13. In addition to the usual interface places, the template also contains the auxiliary place $\mathit{FailSafe}_v$. A token in $\mathit{FailSafe}_v$ indicates that v has become fail-safe due to a failure violating the failure ordering. A token can be placed in $\mathit{FailSafe}_v$ by one of

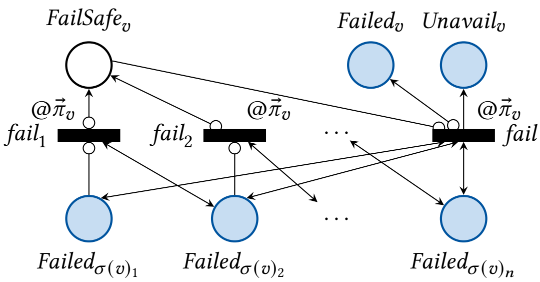


Figure 3.13: GSPN template for $\text{PAND}_{\leq}$

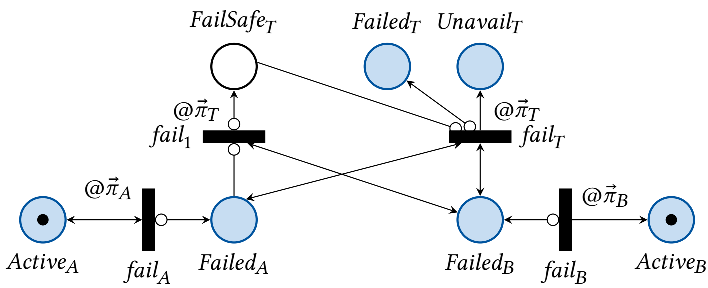


Figure 3.14: GSPN representing a $\text{PAND}_{\leq}$ with two BEs

the transitions $fail_1, \dots, fail_{n-1}$. A transition $fail_i$ fires if child $\sigma(v)_{i+1}$ has failed but child $\sigma(v)_i$ is still operational, i.e. the failures did not happen in the correct order. If failures of two neighbouring children $\sigma(v)_i$ and $\sigma(v)_{i+1}$ occur simultaneously, both places $Failed_{\sigma(v)_i}$ and $Failed_{\sigma(v)_{i+1}}$ contain a token. In that case, the inhibitor arc of $fail_i$ prevents the transition from firing and the gate is not fail-safe. Transition *fail* can fire if all children have failed and no token is in $FailSafe_v$, i.e., the $\text{PAND}_{\leq}$ is not fail-safe.

Note that the priorities $\vec{\pi}_v$ of the GSPN template play a crucial role for the correct handling of failures. The priorities define the order in which transitions are enabled and therefore represent the evaluation order in the DFT.

Example 3.5 (Priorities in PAND template) Consider the (simplified) GSPN in Figure 3.14. The GSPN represents a $\text{PAND}_{\leq}$ T with two BEs A and B as children. For sake of presentation, both BEs fail with immediate transitions instead of timed transitions. We have three different priorities $\vec{\pi}_T$, $\vec{\pi}_A$ and $\vec{\pi}_B$ in the GSPN.

First, we assume the priority ordering $\vec{\pi}_B > \vec{\pi}_A > \vec{\pi}_T$. Both transitions fail_A and fail_B are in concession. Due to the priorities, fail_B fires first and places a token in Failed_B. Afterwards, transitions fail_A and fail_1 are in concession. Note that fail_1 would render T fail-safe. Due to the priorities, transition fail_A fires and places a token in Failed_A. As a result, fail_1 is not in concession any more but fail_T is. Transition fail_T fires and indicates the failure of T.
Because of the priorities, we evaluate $\mathsf{PAND}_{\leq}$ T only after both BEs have failed. Therefore, in this instance, it cannot be detected that Failed_B contained a token before Failed_A. At the point of evaluation of $\mathsf{PAND}_{\leq}$ T, both BEs are failed. The given GSPN priorities represent the trace $\mathrm{f}_B\ \mathrm{f}_A\ \mathrm{f}_T$.
We now consider a different priority ordering $\vec{\pi}_T > \vec{\pi}_B > \vec{\pi}_A$. Transition fail_B again fires first. However, due to the higher priorities, transition fail_1 fires next and places a token in $\mathit{FailSafe}_T$. Next, fail_A fires and no more transitions are enabled. In this instance, $\mathsf{PAND}_{\leq}$ T becomes fail-safe instead of failed. The corresponding trace is $\mathrm{f}_B \oslash_T \mathrm{f}_A \oslash_T$.

Correctness of GSPN template | We prove Theorem 3.1 for $\mathsf{templ}_{\mathsf{PAND}_{\leq}}(v)$:

$$\forall v \in \mathsf{PAND}_{\leq} : \forall \tau \in \mathsf{Traces} :$$
$$v \in \mathsf{Failed}(\tau) \iff \mathsf{m}(\tau)(\mathit{Failed}_v) > 0.$$

Proof. It suffices to show Equation (3.1) on page 62:

$$\mathsf{Eval}(\tau', v) = \mathrm{f}_v \iff \exists t_1, \ldots, t_n \in \mathsf{templ}_{\mathit{Type}(v)}(v) : \mathsf{m}(\tau') \xrightarrow{t_1} \ldots \xrightarrow{t_n} \mathsf{m}(\tau)$$
$$\wedge\ \mathsf{m}(\tau)(\mathit{Failed}_v) > 0.$$

We prove both cases.

- If $\mathsf{Eval}(\tau', v) = \mathrm{f}_v$, then by definition it holds

$$\mathsf{Fail}(\tau, v) \implies \bigwedge_{i=1}^{n} \mathrm{f}_{\sigma(v)_i} \in \tau \wedge \bigwedge_{i=2}^{n} \mathrm{f}_{\sigma(v)_i} \not\prec^{\tau}_{v} \mathrm{f}_{\sigma(v)_{i-1}}.$$

  By induction, $\mathsf{m}(\tau)(\mathit{Failed}_{\sigma(v)_i}) > 0$ for all $1 \leq i \leq n$. Thus, transition fail is in concession—if Failed_v and $\mathit{FailSafe}_v$ do not contain a token. Place Failed_v does not contain a token, because $\mathsf{m}(\tau')(\mathit{Failed}_v) = 0$. If $\mathit{FailSafe}_v$ does not contain a token, firing the transition fail leads to $\mathsf{m}(\tau)(\mathit{Failed}_v) > 0$.

  Otherwise, if $\mathit{FailSafe}_v$ contains a token, one of the transitions $\mathit{fail}_1, \ldots, \mathit{fail}_{n-1}$ must have fired before. W.l.o.g. assume fail_1 fired. The transition can only fire if there is a previous marking m' where $\mathsf{templ}_{\mathsf{PAND}_\leq}(v)$ was evaluated. As v was evaluated in m', there must be a corresponding sub-trace $\tau'' \subseteq \tau$ with $v \in \mathsf{NextEval}(\tau'')$ and $\mathsf{m}(\tau'') = m'$. Transition $fail_1$ can only fire in m' if

$$m'(Failed_{\sigma(v)_1}) = 0 \text{ and } m'(Failed_{\sigma(v)_2}) \geq 1.$$

By induction hypothesis it must therefore hold $\mathrm{f}_{\sigma(v)_1} \notin \tau''$ and $\mathrm{f}_{\sigma(v)_2} \in \tau''$. As v is evaluated after τ'' and eventually $\mathrm{f}_{\sigma(v)_1} \in \tau$, the observed trace order contains

$$\mathrm{f}_{\sigma(v)_2} \blacktriangleleft_v^\tau \mathrm{f}_{\sigma(v)_1}.$$

However, this violates the failure behaviour of the $\mathsf{PAND}_\leq$. Thus, $\neg\mathsf{Fail}(\tau, v)$ which is a contradiction and therefore $FailSafe_v$ cannot contain a token.

- If $\mathsf{Eval}(\tau, v) \neq \mathrm{f}_v$, then two cases are possible:
  (1) Either there exists at least one $1 \leq j \leq n$ such that $\mathrm{f}_{\sigma(v)_j} \notin \tau$ or
  (2) there exists $2 \leq j \leq n$ such that $\mathrm{f}_{\sigma(v)_j} \blacktriangleleft_v^\tau \mathrm{f}_{\sigma(v)_{j-1}}$.

  We start with the first case (1). W.l.o.g. we assume $\mathrm{f}_{\sigma(v)_1} \notin \tau$. Then by induction $\mathsf{m}(\tau)(Failed_{\sigma(v)_1}) = 0$ and transition $fail$ is not enabled. Due to the *monotonicity* of failures, $fail$ was also never enabled before. Thus, $\mathsf{m}(\tau)(Failed_v) = 0$.

  We consider the second case (2). W.l.o.g. we assume $\mathrm{f}_{\sigma(v)_2} \blacktriangleleft_v^\tau \mathrm{f}_{\sigma(v)_1}$. By definition of the observed trace order that means there was a previous evaluation of v where $\sigma(v)_2$ is failed but $\sigma(v)_1$ is not. In other words, there exists a sub-trace $\tau'' \subseteq \tau$ with $v \in \mathsf{NextEval}(\tau'')$ where $\mathrm{f}_{\sigma(v)_1} \notin \tau''$ and $\mathrm{f}_{\sigma(v)_2} \in \tau''$. Then by induction $\mathsf{m}(\tau'')(Failed_{\sigma(v)_1}) = 0$ and $\mathsf{m}(\tau'')(Failed_{\sigma(v)_2}) \geq 1$. Therefore, transition $fail_1$ was enabled in $\mathsf{m}(\tau'')$ and firing $fail_1$ placed a token in $FailSafe_v$. Tokens in $FailSafe_v$ are never removed and therefore $\mathsf{m}(\tau)(FailSafe_v) \geq 1$. As a result, transition $fail$ is never enabled and $\mathsf{m}(\tau)(Failed_v) = 0$. ■

3.5.1.2 Exclusive PAND

·exclusive

The *exclusive* PAND-gate $\mathsf{PAND}_<$ behaves similar to $\mathsf{PAND}_\leq$ but does not allow simultaneous failures.

Definition 3.9 (Failure of $\mathsf{PAND}_<$) Let $v \in \mathsf{PAND}_<$ and $\tau \in \mathsf{Traces}$.

$$\mathsf{Fail}(\tau, v) \iff \bigwedge_{i=1}^{n} \mathrm{f}_{\sigma(v)_i} \in \tau \wedge \bigwedge_{i=1}^{n-1} \mathrm{f}_{\sigma(v)_i} \blacktriangleleft_v^\tau \mathrm{f}_{\sigma(v)_{i+1}}.$$

The failure behaviour of $\mathsf{PAND}_<$ consists of the failure behaviour of AND extended with the constraint ensuring that each child fails strictly before its right sibling. Note that the first constraint is implied by the second constraint, because $e \blacktriangleleft^{\tau}_{v} e' \implies e, e' \in \tau$. From now on, we use the simplified version for the failure of $\mathsf{PAND}_{<}\ v$:

$$\mathsf{Fail}(\tau, v) \iff \bigwedge_{i=1}^{n-1} \mathrm{f}_{\sigma(v)_i} \blacktriangleleft^{\tau}_{v} \mathrm{f}_{\sigma(v)_{i+1}}.$$

All failures in $\mathsf{PAND}_{<}$ are also failures in $\mathsf{PAND}_{\leq}$.

Lemma 3.5 ($\mathsf{PAND}_{<}$ is subset of $\mathsf{PAND}_{\leq}$ w.r.t. failures) Let $v \in \mathsf{PAND}_{<}$, $v' \in \mathsf{PAND}_{\leq}$ and $\tau \in \mathsf{Traces}$.

$$\mathsf{Fail}(\tau, v) \implies \mathsf{Fail}(\tau, v').$$

Proof.

$$\begin{aligned}
\mathsf{Fail}(\tau, v) &\iff \bigwedge_{i=1}^{n-1} \mathrm{f}_{\sigma(v)_i} \blacktriangleleft^{\tau}_{v} \mathrm{f}_{\sigma(v)_{i+1}} \\
&\implies \bigwedge_{i=1}^{n} \mathrm{f}_{\sigma(v)_i} \in \tau \wedge \bigwedge_{i=1}^{n-1} \mathrm{f}_{\sigma(v)_i} \blacktriangleleft^{\tau}_{v} \mathrm{f}_{\sigma(v)_{i+1}} \\
&\implies \bigwedge_{i=1}^{n} \mathrm{f}_{\sigma(v')_i} \in \tau \wedge \bigwedge_{i=2}^{n} \mathrm{f}_{\sigma(v')_{i-1}} \blacktriangleleft^{\tau}_{v} \mathrm{f}_{\sigma(v')_i} \\
&\implies \bigwedge_{i=1}^{n} \mathrm{f}_{\sigma(v')_i} \in \tau \wedge \bigwedge_{i=2}^{n} \mathrm{f}_{\sigma(v')_i} \not\blacktriangleleft^{\tau}_{v'} \mathrm{f}_{\sigma(v')_{i-1}} \\
&\iff \mathsf{Fail}(\tau, v')
\end{aligned}$$

The last implication holds because $\blacktriangleleft^{\tau}_{v}$ is a *strict partial order*. ∎

The template $\mathsf{templ}_{\mathsf{PAND}_{<}}(v)$ for an exclusive PAND with n children is depicted in Figure 3.15. The template for $\mathsf{PAND}_{<}$ is specifically tailored to prevent simultaneous failures and therefore does only share slight resemblance to $\mathsf{templ}_{\mathsf{PAND}_{\leq}}(v)$. The template $\mathsf{templ}_{\mathsf{PAND}_{<}}(v)$ contains auxiliary places $X_1, \ldots, X_{n-1}$. A token in X_i represents that all failures up to child $\sigma(v)_i$ happened in the correct order. Transition fail_i fires and places a token in X_i if the corresponding child $\sigma(v)_i$ has failed, child $\sigma(v)_{i+1}$ is still operational and all previous failures also satisfied the order requirement, i.e. X_{i-1} contains a token as well. The inhibitor arc to the next child $\sigma(v)_{i+1}$ disables fail_i in case of simultaneous failures of $\sigma(v)_i$ and $\sigma(v)_{i+1}$. The first transition fail_1 has no previous place X_0 and the last transition fail_n uses the place Failed_v as place X_n.

Note that the given template is slightly different from the original template in [4].

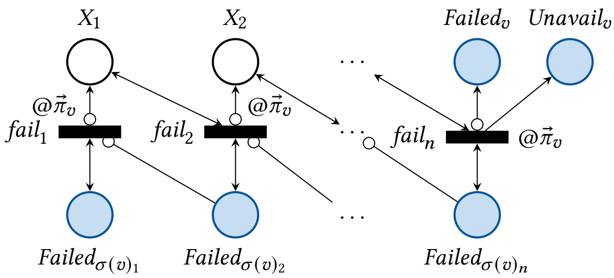


Figure 3.15: GSPN template for $\mathsf{PAND}_<$

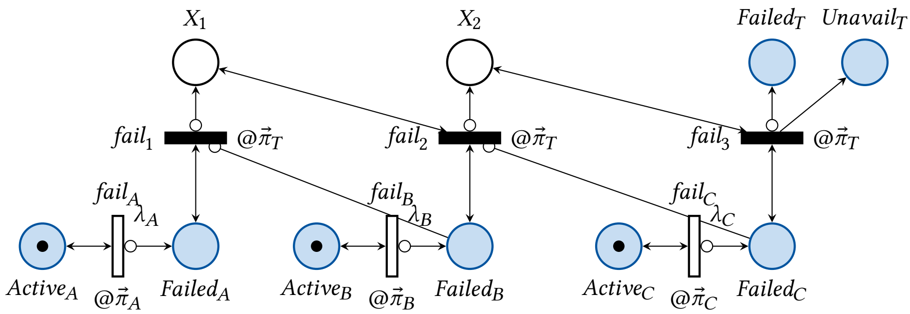


Figure 3.16: GSPN representing a $\mathsf{PAND}_<$ with three BEs

In the original template, firing $fail_i$ moves the token from X_{i-1} to X_i whereas in the template given here, the token is also kept in place X_{i-1}. Keeping the tokens simplifies the correctness proof, but is otherwise equivalent to the original template of [4].

In contrast to $\mathsf{templ}_{\mathsf{PAND}_\leq}(v)$, no dedicated place $FailSafe_v$ is present in this template. If a failure happens out of order, the inhibitor arc from $Failed_{\sigma(v)_i}$ disables transition $fail_{i-1}$. As a result, the token cannot move further than X_{i-1} and never reaches $Failed_v$.

Example 3.6 ($\mathbf{PAND}_<$) Consider the (simplified) GSPN in Figure 3.16. The GSPN represents a $\mathsf{PAND}_<$ T with three BEs A, B and C as children. The priority ordering is $\vec{\pi}_A > \vec{\pi}_B > \vec{\pi}_C > \vec{\pi}_T$.
We consider the ftrace $\mathbf{f}_A\ \mathbf{f}_B\ \mathbf{f}_C$. In the GSPN, first transition $fail_A$ fires and places a token in $Failed_A$. As a result, transition $fail_1$ can fire and places a token in X_1. Next, a token is placed in $Failed_B$, transition $fail_2$ fires and places a token in X_2. The token in X_2 indicates that the failures of the first two children satisfied the ordering requirement. Last, a token is placed in $Failed_C$ and transition $fail_3$ is

enabled. Firing the transition puts a token in Failed_T indicating the failure of $\mathsf{PAND}_<\ T$.
We also consider the ftrace $\mathbf{f}_A\ \mathbf{f}_C\ \mathbf{f}_B$ which violates the order requirement. In the GSPN, at first, tokens are placed in Failed_A and X_1. Next, a token is put in Failed_C and no further immediate transition is enabled. The failure of B places a token in Failed_B. The transition fail_2 however is not enabled because of the inhibitor arc from Failed_C and the token in this place. Thus, $\mathsf{PAND}_<\ T$ cannot fail any more and has become fail-safe.

Correctness of GSPN template | We first prove that the auxiliary place X_i contains a token iff the first i children failed in the correct order.

Lemma 3.6 (X_i places) Let $v \in \mathsf{PAND}_<$, $\tau \in \mathsf{Traces}$ and $1 \leq i \leq |\sigma(v)| - 1$. Then the following holds:

$$\mathsf{m}(\tau)(X_i) > 0 \iff \bigwedge_{j=1}^{i-1} \mathrm{f}_{\sigma(v)_j} \blacktriangleleft_v^\tau \mathrm{f}_{\sigma(v)_{j+1}} \land \left(\mathrm{f}_{\sigma(v)_{i+1}} \notin \tau \lor \mathrm{f}_{\sigma(v)_i} \blacktriangleleft_v^\tau \mathrm{f}_{\sigma(v)_{i+1}}\right).$$

Proof. We perform an induction over the length of the trace τ.

- $\tau = \varepsilon$: Then $\mathrm{f}_{\sigma(v)_1} \notin \tau$ and the right-hand side is false for all $1 \leq i \leq |\sigma(v)| - 1$. By definition of the template $\mathsf{templ}_{\mathsf{PAND}_<}(v)$, it also holds $\mathsf{m}(\varepsilon)(X_i) = 0$ for all i.
- $\tau = \tau' e$: Let $\tau'' \subseteq \tau'$ be the trace after which v was evaluated last, i.e. $\tau' = \tau'' e_v \tau'''$ with $e_v \in \mathsf{Events}^v$ and $\tau''' \cap \mathsf{Events}^v = \emptyset$. We distinguish the following cases for the constraints on the right-hand side:
  - The first constraint is not satisfied.

    W.l.o.g. we assume it is violated for $j = i-1$, i.e. $\mathrm{f}_{\sigma(v)_{i-1}} \not\blacktriangleleft_v^\tau \mathrm{f}_{\sigma(v)_i}$.

    We start with the case where either $\mathrm{f}_{\sigma(v)_{i-1}} \notin \tau$ or $\mathrm{f}_{\sigma(v)_i} \notin \tau$. Then the transition fail_{i-1} or fail_i was never enabled and thus, $\mathsf{m}(\tau)(X_i) = 0$.

    In the following, we assume $\mathrm{f}_{\sigma(v)_{i-1}} \in \tau$ and $\mathrm{f}_{\sigma(v)_i} \in \tau$. W.l.o.g. we assume the failure of $\sigma(v)_{i-1}$ happened after τ'', i.e. $\mathrm{f}_{\sigma(v)_{i-1}} \in \tau'''$.

    Then by induction hypothesis $\mathsf{m}(\tau'')(\mathit{Failed}_{\sigma(v)_{i-1}}) = 0$ and

    $$\mathsf{m}(\tau)(\mathit{Failed}_{\sigma(v)_{i-1}}) > 0 \land \mathsf{m}(\tau)(\mathit{Failed}_{\sigma(v)_i}) > 0.$$

    Thus, transition fail_{i-1} was not enabled in $\mathsf{m}(\tau'')$. However, due to the inhibitor arc of fail_{i-1}, the firing is also prevented in $\mathsf{m}(\tau)$. As also no evaluation of v happened in-between, the transition fail_{i-1} was never enabled and no token can be put in X_{i-1}. As a consequence, transition $fail_i$ is also never enabled and $\mathsf{m}(\tau)(X_i) = 0$.

- The first constraint is satisfied, but the second one is not.

  That means
  $$\mathrm{f}_{\sigma(v)_{i+1}} \in \tau \wedge \mathrm{f}_{\sigma(v)_i} \not\blacktriangleleft^{\tau}_{v} \mathrm{f}_{\sigma(v)_{i+1}}.$$
  W.l.o.g. we assume $\mathrm{f}_{\sigma(v)_{i+1}} \in \tau'''$. Then also $\mathrm{f}_{\sigma(v)_i} \in \tau'''$, because otherwise an ordering could be observed. Following the same reasoning as before, transition $fail_i$ is never enabled and thus, $\mathsf{m}(\tau)(X_i) = 0$.

- Both constraints are satisfied.

  If the constraints were already satisfied for τ'', it holds $\mathsf{m}(\tau'')(X_i) > 0$ by induction. Tokens are never removed from X_i and therefore $\mathsf{m}(\tau)(X_i) > 0$.

  It remains to consider the cases where at least one of the constraints was not satisfied for τ''.

  * Assume the first constraint is satisfied in τ''. Then the second constraint must be violated:
    $$\mathrm{f}_{\sigma(v)_{i+1}} \in \tau'' \wedge \mathrm{f}_{\sigma(v)_i} \not\blacktriangleleft^{\tau''}_{v} \mathrm{f}_{\sigma(v)_{i+1}}.$$
    However, as $\sigma(v)_{i+1}$ is failed and $\sigma(v)_i$ has not failed before, for trace τ it also holds $\mathrm{f}_{\sigma(v)_i} \not\blacktriangleleft^{\tau}_{v} \mathrm{f}_{\sigma(v)_{i+1}}$. Thus, the second constraint is also violated in τ, which is a contradiction.
  * Assume the first constraint is violated in τ''. As the constraint must be satisfied in τ it follows $\mathrm{f}_{\sigma(v)_j} \blacktriangleleft^{\tau}_{v} \mathrm{f}_{\sigma(v)_{j+1}}$, but $\mathrm{f}_{\sigma(v)_j} \not\blacktriangleleft^{\tau''}_{v} \mathrm{f}_{\sigma(v)_{j+1}}$ for some $1 \leq j \leq i-1$. That means, the ordering could not be observed after τ'' but can be observed in τ. As e_v was the last point, where an evaluation took place, it must hold $\mathrm{f}_{\sigma(v)_j} \in \tau''$ and $\mathrm{f}_{\sigma(v)_{j+1}} \in \tau'''$. As the first constraint must be satisfied in τ, only one child can fail in τ'''. Furthermore, it must be the last child $\sigma(v)_i$ which fails in τ'''. That means, for all children $\sigma(v)_j$ with $j < i$ the constraints were satisfied in τ''. As the children failed before in the correct order, it must hold by induction $\mathsf{m}(\tau)(X_{i-1}) > 0$. Failure $\mathrm{f}_{\sigma(v)_i} \in \tau'''$ puts a token in $Failed_{\sigma(v)_i}$. Because of the second constraint, it must hold $\mathrm{f}_{\sigma(v)_{i+1}} \notin \tau$ and therefore $\mathsf{m}(\tau)(Failed_{\sigma(v)_{i+1}}) = 0$. Thus, transition $fail_i$ is enabled and firing it results in $\mathsf{m}(\tau)(X_i) > 0$. ■

Using Lemma 3.6 we prove Theorem 3.1 for $\mathsf{templ}_{\mathsf{PAND}_<}(v)$:

$$\forall v \in \mathsf{PAND}_< : \forall \tau \in \mathsf{Traces} : \\ v \in \mathsf{Failed}(\tau) \iff \mathsf{m}(\tau)(\mathit{Failed}_v) > 0.$$

Proof. The following holds because of Definition 3.9 and Lemma 3.6:

$$\begin{aligned}
\mathsf{Fail}(\tau, v) &\iff \bigwedge_{i=1}^{n-1} \mathsf{f}_{\sigma(v)_i} \blacktriangleleft_v^\tau \mathsf{f}_{\sigma(v)_{i+1}} \\
&\iff \bigwedge_{i=1}^{n} \mathsf{f}_{\sigma(v)_i} \in \tau \wedge \bigwedge_{i=1}^{n-1} \mathsf{f}_{\sigma(v)_i} \blacktriangleleft_v^\tau \mathsf{f}_{\sigma(v)_{i+1}} \\
&\iff \left(\bigwedge_{j=1}^{n-2} \mathsf{f}_{\sigma(v)_j} \blacktriangleleft_v^\tau \mathsf{f}_{\sigma(v)_{j+1}} \wedge \mathsf{f}_{\sigma(v)_{n-1}} \blacktriangleleft_v^\tau \mathsf{f}_{\sigma(v)_n} \right) \wedge \mathsf{f}_{\sigma(v)_n} \in \tau \\
&\iff \mathsf{m}(\tau)(X_{n-1}) > 0 \wedge \mathsf{m}(\tau)(\mathit{Failed}_{\sigma(v)_n})
\end{aligned}$$

It suffices to show Equation (3.1) on page 62:

$$\begin{aligned}
\mathsf{Eval}(\tau', v) = \mathsf{f}_v \iff \exists t_1, \ldots, t_n \in \mathsf{templ}_{\mathit{Type}(v)}(v) : \mathsf{m}(\tau') \xrightarrow{t_1} \ldots \xrightarrow{t_n} \mathsf{m}(\tau) \\
\wedge\, \mathsf{m}(\tau)(\mathit{Failed}_v) > 0.
\end{aligned}$$

We prove both cases.

- If $\mathsf{Eval}(\tau, v) = \mathsf{f}_v$, then by definition $\mathsf{Fail}(\tau, v)$ holds. That means, both places X_{n-1} and $\mathit{Failed}_{\sigma(v)_n}$ contain a token. Thus, transition fail_n can fire and $\mathsf{m}(\tau)(\mathit{Failed}_v) > 0$.
- If $\mathsf{Eval}(\tau, v) \neq \mathsf{f}_v$, then $\mathsf{Fail}(\tau, v)$ does not hold. Then either $\mathsf{f}_{\sigma(v)_n} \notin \tau$ holds—and $\mathit{Failed}_{\sigma(v)_n}$ contains no token—or X_{n-1} contains no token. In both cases, transition fail_n is not enabled. As a result, $\mathsf{m}(\tau)(\mathit{Failed}_v) = 0$. ■

3.5.2  Priority OR (POR)

The priority OR-gate (POR) fails if the first child fails before all other children. It is therefore an extension of the OR-gate by incorporating an ordering requirement and behaves similar to the PAND-gate. As with the PAND, we distinguish inclusive and exclusive variants of the POR.

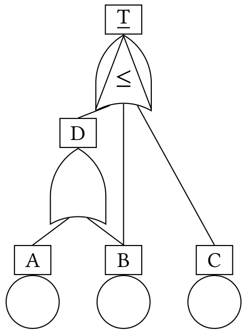


Figure 3.17: Example DFT with inclusive $\mathsf{POR}_{\leq}$

3.5.2.1 Inclusive POR

·inclusive

The *inclusive* POR-gate ($\mathsf{POR}_{\leq}$) allows simultaneous failures of the first child and other children.

Definition 3.10 (Failure of $\mathsf{POR}_{\leq}$) Let $v \in \mathsf{POR}_{\leq}$ and $\tau \in \mathsf{Traces}$.

$$\mathsf{Fail}(\tau, v) \iff \mathrm{f}_{\sigma(v)_1} \in \tau \wedge \bigwedge_{i=2}^{n} \mathrm{f}_{\sigma(v)_i} \not\prec^{\tau}_{v} \mathrm{f}_{\sigma(v)_1}.$$

The failure behaviour of $\mathsf{POR}_{\leq}$ is a combination of the failure behaviour of a standard OR-gate and the ordering requirement. The OR-gate fails if at least one of the children fails and the ordering requires that the child which fails first always is the left-most child. The first part of the failure behaviour can therefore be simplified from $\bigvee_{i=1}^{n} \mathrm{f}_{\sigma(v)_i} \in \tau$ to $\mathrm{f}_{\sigma(v)_1} \in \tau$. Note that the second part which ensures the correct failure order only restricts the ordering w.r.t. to the first child. After the first child has failed, the order in which the other children fail is irrelevant. This is different to the PAND which requires the correct failure order for all children.

Example 3.7 (POR) Consider the DFT with a $\mathsf{POR}_{\leq}$-gate in Figure 3.17. We consider the ftrace $\boldsymbol{\tau}_{\mathrm{f}} = \mathbf{f}_B\, \mathbf{f}_C$. One possible trace $\tau \in \mathsf{Traces}(\boldsymbol{\tau}_{\mathrm{f}})$ is

$$\tau = \mathbf{f}_B\, \mathrm{f}_D\, \mathrm{f}_T\, \mathbf{f}_C.$$

In τ, B fails first and simultaneously D fails as well. As T is an inclusive $\mathsf{POR}_{\leq}$, the simultaneous failure of the first and second child renders the $\mathsf{POR}_{\leq}$ failed. The subsequent failure of C has no impact since T already failed.

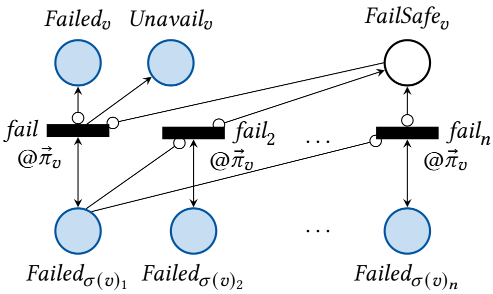


Figure 3.18: GSPN template for $\mathsf{POR}_{\leq}$

In contrast, consider ftrace $\tau'_{\mathsf{f}} = \mathsf{f}_C\, \mathsf{f}_A$. A corresponding trace $\tau' \in \mathsf{Traces}(\tau'_{\mathsf{f}})$ is

$$\tau' = \mathsf{f}_C \oslash_T \mathsf{f}_A\, \mathsf{f}_D \oslash_T.$$

As C fails before the first child, the $\mathsf{POR}_{\leq}$ becomes fail-safe.

The template $\mathsf{templ}_{\mathsf{POR}_{\leq}}(v)$ for an inclusive POR with n children is given in Figure 3.18. The template has a dedicated auxiliary place $FailSafe_v$. A token is placed in $FailSafe_v$ if one of the children $\sigma(v)_2, \dots, \sigma(v)_n$ fails strictly before the first child and therefore violates the order requirement. In that case, transition $fail_i$ is enabled, since $Failed_{\sigma(v)_i}$ contains a token whereas $Failed_{\sigma(v)_1}$ does not. Firing $fail_i$ places a token in $FailSafe_v$ and disables the transition $fail$. Transition $fail$ represents the failure of $\mathsf{POR}_{\leq}$ v. The transition is enabled if the first child is failed, i.e. there is a token in $Failed_{\sigma(v)_1}$, and the $\mathsf{POR}_{\leq}$ is not fail-safe yet, i.e. there is no token in $FailSafe_v$. Firing $fail$ places a token in $Failed_v$ indicating the failure of the $\mathsf{POR}_{\leq}$.

Correctness of GSPN template | We prove Theorem 3.1 for $\mathsf{templ}_{\mathsf{POR}_{\leq}}(v)$:

$$\forall v \in \mathsf{POR}_{\leq} : \forall \tau \in \mathsf{Traces} :$$
$$v \in \mathsf{Failed}(\tau) \iff \mathsf{m}(\tau)(Failed_v) > 0.$$

Proof (sketch). The proof can be constructed similar to the proof for $\mathsf{PAND}_{\leq}$. ∎

3.5.2.2 Exclusive POR

The *exclusive* POR-gate ($\mathsf{POR}_{<}$) prohibits simultaneous failures and requires the first child to fail strictly before all other children. ·exclusive

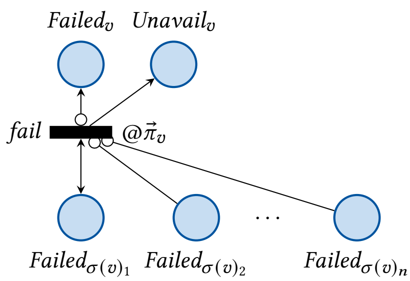


Figure 3.19: GSPN template for $\mathrm{POR}_<$

Definition 3.11 (Failure of $\mathrm{POR}_<$) Let $v \in \mathrm{POR}_<$ and $\tau \in \mathrm{Traces}$.

$$\mathsf{Fail}(\tau, v) \iff \mathrm{f}_{\sigma(v)_1} \in \tau \wedge \bigwedge_{i=2}^{n} \left(\mathrm{f}_{\sigma(v)_i} \notin \tau \vee \mathrm{f}_{\sigma(v)_1} \blacktriangleleft_v^\tau \mathrm{f}_{\sigma(v)_i}\right).$$

The definition extends the failure behaviour of OR with a constraint similar to the second constraint of $\mathrm{PAND}_<$ (cf. Definition 3.9 on page 78). Events are only part of the observed trace order if they occur in the trace. Thus, we also add the case $\mathrm{f}_{\sigma(v)_i} \notin \tau$.

Note that the second constraint from the failure behaviour of $\mathrm{POR}_\leq$ is implied by the second constraint from $\mathrm{POR}_<$:

$$\begin{aligned}\bigwedge_{i=2}^{n} \left(\mathrm{f}_{\sigma(v)_i} \notin \tau \vee \mathrm{f}_{\sigma(v)_1} \blacktriangleleft_v^\tau \mathrm{f}_{\sigma(v)_i}\right) &\implies \bigwedge_{i=2}^{n} \left(\mathrm{f}_{\sigma(v)_i} \notin \tau \vee \mathrm{f}_{\sigma(v)_i} \not\blacktriangleleft_v^\tau \mathrm{f}_{\sigma(v)_1}\right) \\ &\iff \bigwedge_{i=2}^{n} \mathrm{f}_{\sigma(v)_i} \not\blacktriangleleft_v^\tau \mathrm{f}_{\sigma(v)_1}.\end{aligned}$$

The implication again holds because $\blacktriangleleft_v^\tau$ is a *strict partial order*.

The template $\mathsf{templ}_{\mathrm{POR}_<}(v)$ for an exclusive POR with n children is given in Figure 3.19. Similar to the template for $\mathrm{PAND}_<$, $\mathsf{templ}_{\mathrm{POR}_<}(v)$ does not define a dedicated $FailSafe_v$ place. The template has only one transition *fail* which represents the failure of $\mathrm{POR}_<$. Transition *fail* is enabled if the first child is failed, i.e. $Failed_{\sigma(v)_1}$ contains a token, and all other children are not failed yet, i.e. $Failed_{\sigma(v)_i}$ does not contain a token for $i > 1$. In that case, *fail* can fire and places a token in $Failed_v$.

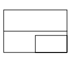
Correctness of GSPN template | We prove Theorem 3.1 for $\mathsf{templ}_{\mathsf{POR}_<}(v)$:

$$\forall v \in \mathsf{POR}_< : \forall \tau \in \mathsf{Traces} :$$
$$v \in \mathsf{Failed}(\tau) \iff \mathsf{m}(\tau)(\mathit{Failed}_v) > 0.$$

Proof. We consider both cases:

- If $\mathsf{m}(\tau)(\mathit{Failed}_v) > 0$, then for some previous marking m transition *fail* fired and placed a token in Failed_v. Then the following must hold: $m(\mathit{Failed}_{\sigma(v)_1}) > 0$ and $m(\mathit{Failed}_{\sigma(v)_i}) = 0$ for $i = 2, \ldots, n$. Let $\tau' \subseteq \tau$ be the corresponding trace for marking m, i.e. $m = \mathsf{m}(\tau')$. By induction it follows that $\mathsf{f}_{\sigma(v)_1} \in \tau'$ and $\mathsf{f}_{\sigma(v)_i} \notin \tau'$ for $i = 2, \ldots, n$. The observed trace order is therefore $\mathsf{f}_{\sigma(v)_1} \blacktriangleleft^{\tau}_{v} \mathsf{f}_{\sigma(v)_i}$ for all children $\sigma(v)_i \in \tau$. Thus, both constraints of Definition 3.11 are satisfied and $\mathsf{Fail}(\tau, v)$ holds. Thus, $\mathsf{f}_v \in \tau$.

- If $\mathsf{m}(\tau)(\mathit{Failed}_v) = 0$, then transition *fail* was never enabled in all previous markings. Either the input arc was never satisfied or one of the inhibitor arcs was violated.
  - The input arc was never satisfied. Then $\mathsf{m}(\tau')(\mathit{Failed}_{\sigma(v)_1}) = 0$ for all traces $\tau' \subseteq \tau$. By induction it follows that $\mathsf{f}_{\sigma(v)_1} \notin \tau'$ and therefore the first constraint of Definition 3.11 was never satisfied. Thus, $\mathsf{f}_v \notin \tau$.
  - The input arc was satisfied at some point. Let $\tau' \subseteq \tau$ be the first trace where v was evaluated and the input arc was satisfied, i.e. $\mathsf{m}(\tau')(\mathit{Failed}_{\sigma(v)_1}) > 0$ and $v \in \mathsf{NextEval}(\tau')$. At the same time, one of the inhibitor arcs must be violated. Otherwise *fail* could have fired. W.l.o.g. we assume the inhibitor arc of the second child was violated, i.e. $\mathsf{m}(\tau')(\mathit{Failed}_{\sigma(v)_2}) > 0$. By induction it follows $\mathsf{f}_{\sigma(v)_1} \in \tau'$ but also $\mathsf{f}_{\sigma(v)_2} \in \tau'$. Moreover, for all previous evaluations $v \in \mathsf{NextEval}(\tau'')$, $\tau'' \subseteq \tau'$, by assumption it holds $\mathsf{f}_{\sigma(v)_1} \notin \tau''$. Thus, $\mathsf{f}_{\sigma(v)_1} \not\blacktriangleleft^{\tau'}_{v} \mathsf{f}_{\sigma(v)_2}$. The second constraint of Definition 3.11 can never be satisfied and $\mathsf{f}_v \notin \tau$. ■

3.6  Spare Gate

The SPARE gate [DBB90; BCS10] is possibly the most complex of all the DFT gates. It is usually used to model redundancies in a system. Besides the usual failure propagation, SPARE gates introduce two new concepts: claiming and activation. We will shortly introduce both concepts informally before giving precise semantics.

Upon failure of the main component, a SPARE switches to a—previously unused—spare component and uses it henceforward. This process is called *claiming*. During

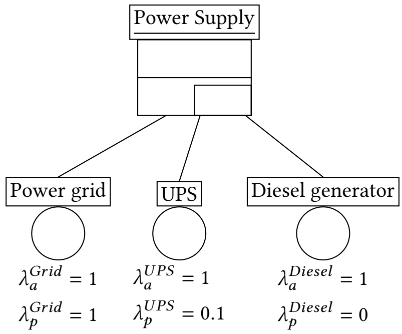


Figure 3.20: Example DFT with SPARE

claiming, the newly used spare component is *activated* and its failure probability might increase. If it is not used, a spare component is in passive mode. A *passive* component might fail with lower probability compared to *active* use or might not even fail at all.

A spare component can be associated with multiple SPARE gates, i.e. it is *shared*. A shared spare component can replace a component in any of the associated SPARE gates. However, claiming is mutual exclusive and only one of the SPARE gates can use the shared spare. Therefore, a SPARE gate cannot claim a spare component if the spare component is already claimed by another SPARE.

Example 3.8 (SPARE) Consider the DFT in Figure 3.20. It models a redundant power supply *PS* by using a SPARE gate. In the beginning the power comes from the power grid *Grid*. Redundant options are the uninterruptible power supply *UPS* and a diesel generator *Diesel*. We consider the trace

$$\tau = \uparrow_{PS} \mathrm{cl}^{PS}_{Grid} \uparrow_{Grid} \mathbf{f}_{Grid} \mathrm{cl}^{PS}_{UPS} \uparrow_{UPS} \mathbf{f}_{UPS} \mathrm{cl}^{PS}_{Diesel} \uparrow_{Diesel} \mathbf{f}_{Diesel} \mathrm{f}_{PS}.$$

In the beginning, the SPARE *PS* is activated, indicated by event $\uparrow_{PS}$. Next, the SPARE claims its first child, the primary component *Grid* as indicated by the event cl^{PS}_{Grid}, and activates it. After the failure of *Grid*, the SPARE switches to the next component *UPS*, i.e. event cl^{PS}_{UPS} happens. Claiming also activates *UPS* yielding a higher failure rate $\lambda_a^{UPS} = 1$ instead of the passive failure rate $\lambda_p^{UPS} = 0.1$ from before. After the failure of *UPS*, the diesel generator is switched on in event $\mathrm{cl}^{PS}_{Diesel}$. Note that the diesel generator has a passive failure rate $\lambda_p^{Diesel} = 0$ and therefore does not fail while not being in use. Last, after the failure of *Diesel*, no spare component is available any more and therefore the SPARE gate fails as well. In trace τ, all BEs failed while being actively used. However, it is also possible to

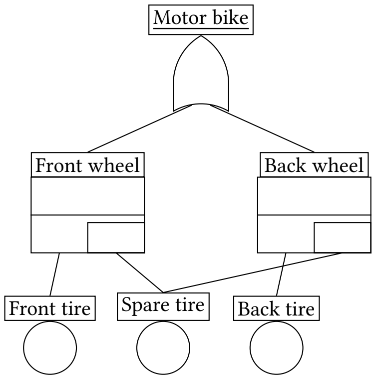


Figure 3.21: Example DFT with shared spare component

have a trace

$$\tau' = \uparrow_{PS} \mathrm{cl}^{PS}_{Grid} \uparrow_{Grid} \mathbf{f}_{UPS} \mathbf{f}_{Grid} \mathrm{cl}^{PS}_{Diesel} \uparrow_{Diesel} \mathbf{f}_{Diesel} \mathrm{f}_{PS}.$$

In τ', UPS fails first while being in passive mode. After the failure of $Grid$, the second child UPS is not available any more due to failure. Therefore, the third child $Diesel$ is immediately claimed and activated.

We consider a second example of SPAREs with a shared spare component.

Example 3.9 (Shared spare components) The DFT in Figure 3.21 models a motor bike $Bike$ which fails if either the front wheel FW or the back wheel BW fails. Both wheels use a corresponding tire, but can also use the spare tire ST if a failure occurs. One possible trace of the DFT is

$$\tau = \uparrow_{Bike} \uparrow_{FW} \uparrow_{BW} \mathrm{cl}^{FW}_{FT} \mathrm{cl}^{BW}_{BT} \uparrow_{FT} \uparrow_{BT} \mathbf{f}_{FT} \mathrm{cl}^{FT}_{ST} \uparrow_{ST} \mathbf{f}_{BT} \mathrm{f}_{BW} \mathrm{f}_{Bike}.$$

First, both SPAREs claim their primary component as indicated by the events cl^{FW}_{FT} for the front wheel and cl^{BW}_{BT} for the back wheel. Both tires are also activated. Next, a failure of the front tire FT occurs. As a result, the front wheel requires the spare tire ST and the claiming event cl^{FT}_{ST} and activation $\uparrow_{ST}$ take place. Next, the back tire BT fails and the back wheel BW requires a new tire as well. However, no spare component is available any more, because ST is already claimed by FW. As a result, the SPARE BW fails and therefore the motor bike fails.

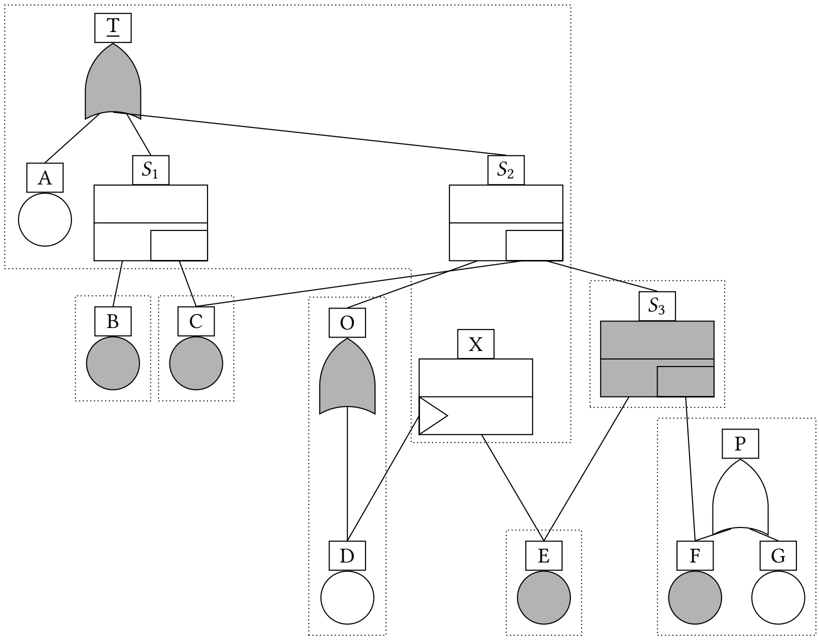


Figure 3.22: Modules (boxes) and module representatives (grey)

3.6.1 Spare modules

·primary child
·spare children

Before giving the precise semantics of the SPARE gate, we fix some notation. Let $s \in \text{SPARE}_{\mathcal{F}}$ be a SPARE gate with n children. The *primary child* is the first child $\sigma(s)_1$ of the SPARE. The other children $\sigma(s)_2, \ldots, \sigma(s)_n$ are called *spare children*.

Originally, SPAREs could only have BEs as children [DBB92] and most semantics kept this restriction. However, in [BCS10] this restriction was first lifted and complete subtrees were allowed as children of SPAREs. We also allow subtrees and therefore need the concept of *modules* as given in Section 2.2.3.3 on page 27. Modules behave as one entity w.r.t. claiming and activation. Each module has a *module representative*: either the child of a SPARE or the top-level element of the DFT. A module contains all DFT elements connected to the module representative via a path which does not go through SPAREs, PDEPs or SEQs. Thus, the leafs in a module are either BEs or SPARE gates.

The primary child of a SPARE is the representative of the *primary module* and the spare children are the representatives of the *spare modules*.

Example 3.10 (Spare modules) We depict the modules of an example DFT in Figure 3.22. The modules are highlighted by dotted boxes and the module representative is indicated by a grey background colour.
The given DFT has seven modules. The top module has five elements including the representative T and two SPARE gates. A module ends at a SPARE and each child of a spare gate forms it owns module. For example, SPARE S_1 has the primary module with representative B and a spare module consisting of C. Note that the module with C is also a spare module of S_2 as it is a shared spare component.
The module with representative O contains BE D but not the FDEP X. FDEP X is part of the top module. The FDEP also prohibits D and E from being part of the same module. Instead E forms its own primary module for S_3.
The module with representative F shows that a module does not necessarily form a subtree but can also include 'sideways' elements, such as P and G in this example. While the 'sideways' elements do not necessarily influence the failure of the module representative, they might trigger FDEPs which forward the failure into different modules.

3.6.2 Claiming

We start the semantics of SPARE gates by defining the *claiming* mechanism. In the DFT traces, claiming is denoted by the event cl_v^s meaning SPARE s claims child v. We recall that for a trace τ the *set of claimed elements* of SPARE s is given by $\mathsf{Claimed}(\tau)(s)$ (cf. Definition 2.33 on page 41). In the following we use the phrase 'claiming a child' to mean claiming the complete module of the child. ·claiming

When trying to claim, only available children can be claimed.

Definition 3.12 (Available for claiming) Let $\mathcal{F} = (V, \sigma, Type, top, \Theta)$ be a DFT and let $\tau \in \mathsf{Traces}$ be a trace in $\mathcal{F}$. An element $v \in \mathcal{F}$ is *available for claiming* after τ iff it is neither failed nor already claimed by a SPARE. ·available for claiming

$$v \in \mathsf{Available}(\tau) \iff v \notin \mathsf{Failed}(\tau) \land \forall s \in \mathsf{SPARE}_{\mathcal{F}} : v \notin \mathsf{Claimed}(\tau)(s).$$

Corollary 3.7 (Unavailable for claiming is monotonic) Let $\tau, \tau' \in \mathsf{Traces}$, $\tau' \subseteq \tau$ and $v \in \mathcal{F}$.
Then $v \notin \mathsf{Available}(\tau') \implies v \notin \mathsf{Available}(\tau)$.

Proof. We know from Lemma 2.3 that failures are monotonic:

$$v \in \mathsf{Failed}(\tau') \implies v \in \mathsf{Failed}(\tau).$$

Claiming is also monotonic for all $s \in \text{SPARE}$:

$$v \in \mathsf{Claimed}(\tau')(s) \implies v \in \mathsf{Claimed}(\tau)(s). \qquad \blacksquare$$

Thus,

$$\begin{aligned} v \notin \mathsf{Available}(\tau') &\iff v \in \mathsf{Failed}(\tau') \lor \exists s \in \text{SPARE} : v \in \mathsf{Claimed}(\tau')(s) \\ &\implies v \in \mathsf{Failed}(\tau) \lor \exists s \in \text{SPARE} : v \in \mathsf{Claimed}(\tau)(s) \\ &\iff v \notin \mathsf{Available}(\tau). \end{aligned}$$

When trying to claim a new child, the spare children are considered in order from left to right.

·next spare to claim

Definition 3.13 (Next spare to claim) Let $v \in \text{SPARE}_{\mathcal{F}}$ and $\tau \in \mathsf{Traces}$. The *next spare to claim* for v after τ is given by

$$\mathsf{NextSpare}(\tau, v) = \min_j \left\{ \sigma(v)_j \in \mathsf{Available}(\tau) \right\}.$$

Remark 3.1 (Other claiming orders) It is possible to define a different order of claiming by adapting $\mathsf{NextSpare}(\tau, v)$ accordingly. Claiming might be resolved by a uniform distribution or by non-determinism. The latter for example allows to synthesise optimal claiming strategies [MMGN20].

We can now define the behaviour of SPARE gates.

Definition 3.14 (Evaluation of SPARE) Let $\mathcal{F} = (V, \sigma, Type, top, \Theta)$ be a DFT and let $\tau \in \mathsf{Traces}$.
The evaluation function for $v \in \text{SPARE}_{\mathcal{F}}$ is given as follows:

$$\mathsf{Eval}(\tau, v) = \begin{cases} \mathrm{f}_v & \text{if } \mathrm{f}_v \notin \tau \land \mathsf{Fail}(\tau, v) \\ \mathrm{cl}^v_{\sigma(v)_i} & \text{else if } \mathrm{cl}^v_{\sigma(v)_i} \notin \tau \land \mathsf{Claim}(\tau, v, \sigma(v)_i) \\ \uparrow_v & \text{else if } \uparrow_v \notin \tau \land \mathsf{Act}(\tau, v) \\ \oslash_v & \text{otherwise.} \end{cases}$$

Compared to the general definition of the evaluation function in Definition 2.25, SPARE-gates can return the additional claiming event $\mathrm{cl}^v_{\sigma(v)_i}$. The claiming behaviour of SPARE gates is given by the condition $\mathsf{Claim}(\tau, v, v')$.

Definition 3.15 (Claiming of SPARE) Let $v \in \text{SPARE}_{\mathcal{F}}$, $\tau \in \mathsf{Traces}$, $1 \leq i \leq n$.

$$\mathsf{Claim}(\tau, v, \sigma(v)_i) \iff \mathsf{Claimed}(\tau)(v) \subseteq \mathsf{Failed}(\tau) \wedge \sigma(v)_i = \mathsf{NextSpare}(\tau, v).$$

In the beginning—for the empty trace—no child is claimed yet. A child $\sigma(v)_i$ stays claimed if it was claimed before. Note that also failed children will still remain claimed. That way, the claiming history can be reconstructed if necessary. If all claimed children have failed—and in particular the child claimed last—a new spare child needs to be claimed. The SPARE tries to claim a new child according to the given order from left to right. In the beginning, the left-most child will be claimed.

3

Lemma 3.8 (First child will initially be claimed) Let $v \in \mathsf{SPARE}_{\mathcal{F}}$, $\tau \in \mathsf{Traces}$. The primary child of v will initially be claimed, i.e. $\mathsf{Claim}(\varepsilon, v, \sigma(v)_1)$ holds.

Proof. It holds $\sigma(v)_1 \in \mathsf{Available}(\varepsilon)$, because

$$\sigma(v)_1 \notin \mathsf{Failed}(\varepsilon) \text{ and } \forall s \in \mathsf{SPARE}_{\mathcal{F}} : \mathsf{Claimed}(\varepsilon)(s) = \emptyset.$$

As $\sigma(v)_1$ is the left-most child, which is available, it follows $\mathsf{NextSpare}(\varepsilon, v) = \sigma(v)_1$. By definition

$$\mathsf{Claimed}(\varepsilon)(v) = \emptyset \subseteq \emptyset = \mathsf{Failed}(\varepsilon).$$

and therefore $\mathsf{Claim}(\varepsilon, v, \sigma(v)_1)$ holds. ■

A SPARE fails, if the claiming is unsuccessful. In other words, all claimed children have failed and the remaining children are not available for claiming.

Definition 3.16 (Failure of SPARE) Let $v \in \mathsf{SPARE}$ and $\tau \in \mathsf{Traces}$.

$$\mathsf{Fail}(\tau, v) \iff \mathsf{Claimed}(\tau)(v) \subseteq \mathsf{Failed}(\tau) \wedge (\sigma(v) \cap \mathsf{Available}(\tau)) = \emptyset.$$

3.6.3 GSPN template

The template $\mathsf{templ}_{\mathsf{SPARE}}(v)$ for a SPARE with n children is given in Figure 3.23. The template is slightly different from the original template in [4], because transition *child-fail* also returns the token to $Claimed^{v}_{\sigma(v)_i}$. This follows the definition of $\mathsf{Claimed}(\tau)(v)$ where claimed children are never released. Additionally, auxiliary places $Current_i$ are introduced to ensure the correctness of the changed template.

The template contains the interface places $Failed_v$ and $Unavail_v$ for the SPARE v and all its children. Additionally, the auxiliary places $Consider_{\sigma(v)_i}$, $Claimed^{v}_{\sigma(v)_i}$ and $Current_i$ are present for each child. Place $Current_i$ is only needed to ensure that transition *child-fail* only fires once. A token in $Consider_{\sigma(v)_i}$ represents that child $\sigma(v)_i$ is considered next for claiming. A token in $Claimed^{v}_{\sigma(v)_i}$ represents that the corresponding child is claimed by v.

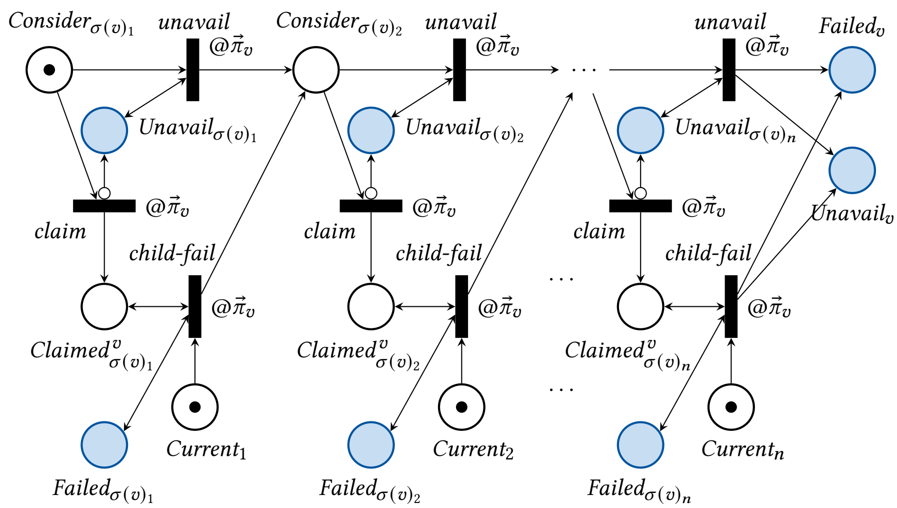


Figure 3.23: GSPN template for SPARE

The current state of the SPARE gate is represented by a 'moving' token in the template. Initially, the token is present in $Consider_{\sigma(v)_1}$ which represents that the primary child is considered for claiming.

Two cases are possible here. If the child is not available for claiming, i.e. a token is present in $Unavail_{\sigma(v)_1}$, the transition *unavail* is enabled. Firing this transition moves the token to $Consider_{\sigma(v)_2}$ and the second child is considered for claiming next.

If the child is available for claiming, no token is present in $Unavail_{\sigma(v)_1}$ and transition *claim* is enabled. Firing the transition moves the token to $Claimed^v_{\sigma(v)_1}$ which indicates that child $\sigma(v)_1$ is claimed now. Additionally, a token is placed in $Unavail_{\sigma(v)_1}$ indicating that the child is not available for claiming by other SPAREs any more.

If a claimed child fails, transition *child-fail* becomes enabled. Firing the transition places a token in $Consider_{\sigma(v)_2}$ indicating that the next child needs to be considered for claiming. The token in $Claimed^v_{\sigma(v)_1}$ is kept as the failed child is still considered claimed. The token in $Current_i$ is consumed to prevent the transition from firing again.

Following these steps, the token moves through all $Consider_{\sigma(v)_i}$ places and finally ends up in the $Failed_v$ place. In that case, all children were considered but none of them are available any more. As a result, the SPARE fails.

Example 3.11 (SPARE template) We consider an example DFT and the corresponding GSPN. The DFT in Figure 3.24 consists of two SPARE gates S and T which

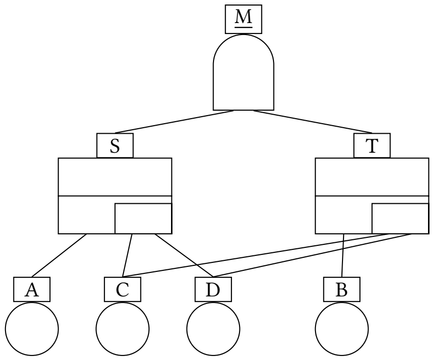


Figure 3.24: DFT with shared spares

share the two BEs C and D. We consider the DFT trace

$$\tau = \mathrm{cl}_A^S\ \mathrm{cl}_B^T\ \mathbf{f}_B\ \mathrm{cl}_C^T\ \mathbf{f}_A\ \mathrm{cl}_D^S\ \mathbf{f}_D\ \mathrm{f}_S.$$

We depict the (partial) GSPN corresponding to SPARE S in Figure 3.25. The marking corresponds to the trace prefix $\mathrm{cl}_A^S \mathrm{cl}_B^T \mathbf{f}_B \mathrm{cl}_C^T$. A token is present in $\mathit{Claimed}_A^S$ because primary A is claimed by S. Additionally, $\mathit{Unavail}_A$ contains a token, because A is now unavailable for the other SPARE T. After the failure of primary B, SPARE T has claimed the first spare child C. A token is present in $\mathit{Unavail}_C$ to indicate that C is unavailable for SPARE S.
The next event in trace τ is the failure of A, i.e. a token will be placed in Failed_A. Transition *child-fail* fires, consumes the token from $\mathit{Current}_1$ and places a token in $\mathit{Consider}_C$. Next, spare child C is considered for claiming. However, as C is already claimed, it is unavailable as indicated by a token in $\mathit{Unavail}_C$. Therefore transition *unavail* fires and moves the token from $\mathit{Consider}_C$ to $\mathit{Consider}_D$. Next, the second spare child D is considered for claiming. As no token is yet present in $\mathit{Unavail}_D$, transition *claim* can fire. As a result, the token is moved to $\mathit{Claimed}_D^S$ indicating that D is now claimed by S. In addition, a token is put into $\mathit{Unavail}_D$ such that T cannot claim it any more.

The $\mathit{Unavail}_v$ interface place indicates whether v is available for claiming. An element is unavailable if it is either failed or already claimed by a SPARE. Due to the second case we cannot use the Failed_v place to indicate unavailability but have to introduce a separate interface place. However, a failure still renders the element unavailable.

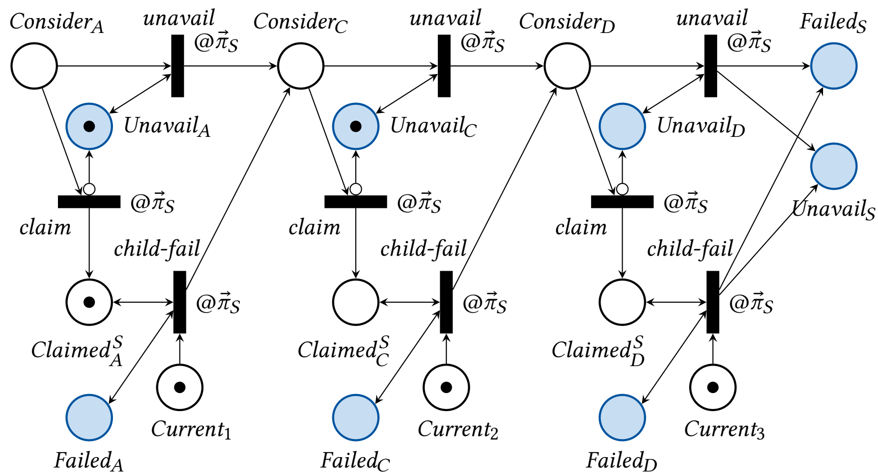


Figure 3.25: GSPN for SPARE S from Figure 3.25

Lemma 3.9 (Failed elements are unavailable for claiming)

$$\forall v \in \mathcal{F} : \forall \tau \in \mathsf{Traces} :$$
$$\mathsf{m}(\tau)(\mathit{Failed}_v) > 0 \implies \mathsf{m}(\tau)(\mathit{Unavail}_v) > 0.$$

Proof. This follows directly from the later Lemma 3.20 on page 130. ∎

Claiming also renders the element unavailable.

Lemma 3.10 (Claimed elements are unavailable for claiming)

$$\forall v \in \mathcal{F} : \forall \tau \in \mathsf{Traces} :$$
$$\exists s \in \mathsf{SPARE}_{\mathcal{F}} : \mathsf{m}(\tau)(\mathit{Claimed}^s_v) > 0 \implies \mathsf{m}(\tau)(\mathit{Unavail}_v) > 0.$$

Proof. Initially, both $\mathit{Claimed}^s_v$ for all SPAREs s and $\mathit{Unavail}_v$ are empty. Transition *claim* is the only transition which can add a new token to $\mathit{Claimed}^s_v$. This transition also simultaneously adds a token to $\mathit{Unavail}_v$—if no token was present before. Thus, if there is one SPARE s with $\mathsf{m}(\tau)(\mathit{Claimed}^s_v) > 0$, then also $\mathsf{m}(\tau)(\mathit{Unavail}_v) > 0$. ∎

Theorem 3.11 ($\mathit{Unavail}_v$ interface place) A token is in interface place

Unavail$_v$ iff the DFT element $v \in \mathcal{F}$ is not *available for claiming*. More formally:

$$\forall v \in \mathcal{F} : \forall \tau \in \mathsf{Traces} :$$
$$v \notin \mathsf{Available}(\tau) \iff \mathsf{m}(\tau)(\mathit{Unavail}_v) > 0.$$

Proof. We prove both directions.

- $\Longrightarrow$: If $v \notin \mathsf{Available}(\tau)$, then by definition either $v \in \mathsf{Failed}(\tau)$ or there exists a SPARE s such that $v \in \mathsf{Claimed}(\tau)(s)$. In the first case, from Theorem 3.1 and Lemma 3.9 we know

  $$v \in \mathsf{Failed}(\tau) \iff \mathsf{m}(\tau)(\mathit{Failed}_v) > 0 \implies \mathsf{m}(\tau)(\mathit{Unavail}_v) > 0.$$

  In the second case, from Theorem 3.12 and Lemma 3.10 we know

  $$v \in \mathsf{Claimed}(\tau)(s) \iff \mathsf{m}(\tau)(\mathit{Claimed}_v^s) > 0 \implies \mathsf{m}(\tau)(\mathit{Unavail}_v) > 0.$$

- $\Longleftarrow$: We show the contraposition. By Lemma 3.9 and Lemma 3.10 it holds:

  $$\mathsf{m}(\tau)(\mathit{Unavail}_v) = 0 \implies \mathsf{m}(\tau)(\mathit{Failed}_v) = 0 \land \forall s \in \mathsf{SPARE} : \mathsf{m}(\tau)(\mathit{Claimed}_v^s) = 0.$$

  With Theorem 3.1 and Theorem 3.12 it follows:

  $$\mathsf{m}(\tau)(\mathit{Failed}_v) = 0 \land \forall s \in \mathsf{SPARE} : \mathsf{m}(\tau)(\mathit{Claimed}_v^s) = 0$$
  $$\iff v \notin \mathsf{Failed}(\tau) \land \forall s \in \mathsf{SPARE} : v \notin \mathsf{Claimed}(\tau)(s)$$

  and thus, $v \in \mathsf{Available}(\tau)$. ■

3.6.3.1 Template for arbitrary claiming

We give the template for a SPARE with arbitrary claiming order in Figure 3.26. The main difference to the SPARE template with claiming from left to right as given in Figure 3.23 is the single place *Consider*$_v$. Instead of moving the token through all *Consider*$_{\sigma(v)_i}$ places one after the other, a token is placed in *Consider*$_v$ whenever a next child should be claimed. In that case, some of the transitions *claim* are enabled depending on which children are available for claiming. After the failure of the claimed child, the token is put back into *Consider*$_v$ and a new child for claiming can be selected. While the child is claimed and operational, no token is present in *Consider*$_v$ and therefore no claiming can take place. If a token is in *Consider*$_v$ but no child is available for claiming any more, transition *unavail* can fire and indicates the failure of the SPARE. Note that if the

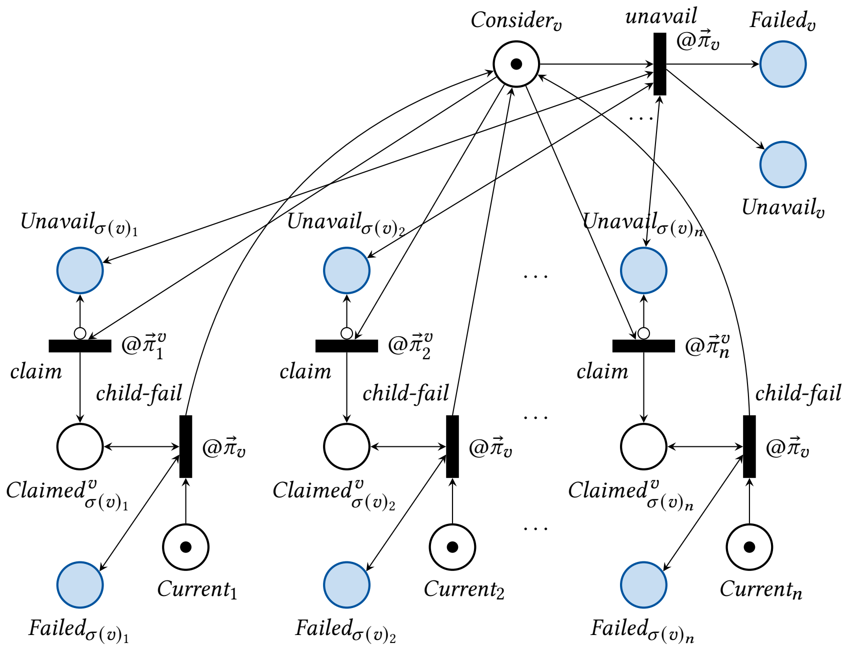


Figure 3.26: GSPN template for SPARE with arbitrary claiming

last child is claimed, all places $Unavail_{\sigma(v)_i}$ contain a token. However, the transition *unavail* is not enabled yet, because the token is not yet in $Consider_v$ but in one of the $Claimed^v_{\sigma(v)_i}$ places.

It is possible that multiple transitions *claim* are enabled simultaneously. We therefore use priority variables $\vec{\pi}^v_1$, ..., $\vec{\pi}^v_n$ which are each associated with one *claim* transition. The priority variables are used to define which transition should be enabled, i.e. which child should be claimed next. By setting the priorities, we can specify the desired claiming order as given by the function $\mathrm{NextSpare}(\tau, v)$.

If we set all priority variables to the same value, the claiming will be resolved either non-deterministically or by random choice—depending on the specified partitioning. We can specify the usual ordering from left to right by using decreasing priorities:

$$\vec{\pi}^v_1 = n, \vec{\pi}^v_2 = n - 1, \dots, \vec{\pi}^v_n = 1.$$

In that instance, only the leftmost transition which has concession is enabled.

3.6.3.2 Correctness of GSPN template

We have now introduced the necessary concepts to prove the correctness of the GSPN template for a SPARE. We consider the template for SPAREs with arbitrary claiming because it is the most general variant and subsumes the claiming from left to right.

3

We start by proving the correctness of the $\mathit{Claimed}^{v}_{v'}$ place.

Theorem 3.12 ($\mathit{Claimed}^{v}_{v'}$ place) A token is in place $\mathit{Claimed}^{v}_{v'}$ of SPARE v iff the DFT element $v' \in \mathcal{F}$ is claimed by v. More formally:

$$\forall v \in \mathsf{SPARE} : \forall v' \in \sigma(v) : \forall \tau \in \mathsf{Traces} :$$
$$v' \in \mathsf{Claimed}(\tau)(v) \iff \mathsf{m}(\tau)(\mathit{Claimed}^{v}_{v'}) > 0.$$

Proof. We perform an induction over trace τ.

- $\tau = \varepsilon$: The initial marking of the GSPN template does not contain any tokens except in $\mathit{Consider}_v$. By definition $\mathsf{Claimed}(\varepsilon)(s) = \emptyset$ for all $s \in \mathsf{SPARE}$. Thus,

$$\sigma(v)_i \notin \mathsf{Claimed}(\varepsilon)(v) \iff \mathsf{m}(\varepsilon)(\mathit{Claimed}^{v}_{\sigma(v)_i}) = 0.$$

- $\tau = \tau'\ e$: We distinguish two cases:
  - We start with the case $e \notin \mathsf{Events}^v$. That means $v \notin \mathsf{NextEval}(\tau')$ and by definition of the GSPN priorities, no transition in $\mathsf{templ}_{\mathsf{SPARE}}(v)$ can fire in $\mathsf{m}(\tau')$. Thus, for all $p \in \left\{\mathit{Claimed}^{v}_{v'} \mid v' \in \sigma(v)\right\}$ it holds:

$$\mathsf{m}(\tau')(p) = \mathsf{m}(\tau)(p).$$

    As $e \neq \mathrm{cl}^{v}_{\sigma(v)_i}$, it follows by induction hypothesis:

$$\begin{aligned} \sigma(v)_i \in \mathsf{Claimed}(\tau)(v) &\iff \sigma(v)_i \in \mathsf{Claimed}(\tau')(v) \\ \overset{I.H.}{\iff} \mathsf{m}(\tau')(\mathit{Claimed}^{v}_{\sigma(v)_i}) > 0 &\iff \mathsf{m}(\tau)(\mathit{Claimed}^{v}_{\sigma(v)_i}) > 0. \end{aligned}$$

  - We consider the case $e = \mathrm{cl}^{v}_{\sigma(v)_i}$. Then $\sigma(v)_i \in \mathsf{Claimed}(\tau)(v)$. It also means $v \in \mathsf{NextEval}(\tau')$ and only transitions in $\mathsf{templ}_{\mathsf{SPARE}}(v)$ can fire in marking

$\mathsf{m}(\tau')$. By Definition 3.14, Definition 3.15 and induction hypothesis it holds:

$$\begin{aligned}
&\mathsf{Eval}(\tau', v) = \mathrm{cl}^v_{\sigma(v)_i}\\
\iff&\mathrm{cl}^v_{\sigma(v)_i} \notin \tau' \wedge \mathsf{Claim}(\tau', v, \sigma(v)_i)\\
\iff&\sigma(v)_i \notin \mathsf{Claimed}(\tau')(v) \wedge \mathsf{Claimed}(\tau')(v) \subseteq \mathsf{Failed}(\tau')\\
&\wedge \sigma(v)_i = \mathsf{NextSpare}(\tau', v)\\
\overset{I.H.}{\iff}&\mathsf{m}(\tau')(\mathit{Claimed}^v_{\sigma(v)_i}) = 0\\
&\wedge \forall \sigma(v)_j : \left(\mathsf{m}(\tau')(\mathit{Claimed}^v_{\sigma(v)_j}) > 0 \implies \mathsf{m}(\tau')(\mathit{Failed}_{\sigma(v)_j}) > 0\right)\\
&\wedge \sigma(v)_i = \mathsf{NextSpare}(\tau', v).
\end{aligned}$$

If we consider the second conjunct, all claimed children failed. That means, one of the transitions *child-fail* can fire and the token is back in $\mathit{Consider}_v$. The next child to claim is $\mathsf{NextSpare}(\tau', v) = \sigma(v)_i$. As $\mathit{Claimed}^v_{\sigma(v)_i}$ does not yet contain a token, transition *claim* for $\sigma(v)_i$ is enabled. Firing the transition yields $\mathsf{m}(\tau)(\mathit{Claimed}^v_{\sigma(v)_i}) > 0$.

It remains to show the other direction. We prove it by contraposition:

$$\mathsf{Eval}(\tau', v) \neq \mathrm{cl}^v_{\sigma(v)_i} \iff \mathrm{cl}^v_{\sigma(v)_i} \in \tau' \vee \neg\mathsf{Claim}(\tau', v, \sigma(v)_i)$$

If the first conjunct $\mathrm{cl}^v_{\sigma(v)_i} \in \tau'$ holds, then we know by induction hypothesis $\mathsf{m}(\tau')(\mathit{Claimed}^v_{\sigma(v)_i}) > 0$. Thus, as before,

$$\sigma(v)_i \in \mathsf{Claimed}(\tau)(v) \implies \mathsf{m}(\tau)(\mathit{Claimed}^v_{\sigma(v)_i}) > 0.$$

We now assume the second conjunct holds and $\mathsf{m}(\tau')(\mathit{Claimed}^v_{\sigma(v)_i}) = 0$.

$$\begin{aligned}
&\neg\mathsf{Claim}(\tau', v, \sigma(v)_i)\\
\iff&\exists \sigma(v)_j : \left(\mathsf{m}(\tau')(\mathit{Claimed}^v_{\sigma(v)_j}) > 0 \wedge \mathsf{m}(\tau')(\mathit{Failed}_{\sigma(v)_j}) = 0\right)\\
&\vee \sigma(v)_i \neq \mathsf{NextSpare}(\tau, v).
\end{aligned}$$

If there is a child $\sigma(v)_j$ which is claimed but not failed, then the token is still in $\mathit{Claimed}^v_{\sigma(v)_j}$ and not in $\mathit{Consider}_v$. Transition *claim* has not fired and

$$\mathsf{m}(\tau)(\mathit{Claimed}^v_{\sigma(v)_i}) = \mathsf{m}(\tau')(\mathit{Claimed}^v_{\sigma(v)_i}) = 0.$$

If all claimed children have failed, then the token is back in $\mathit{Consider}_v$. However, as $\mathsf{NextSpare}(\tau, v) \neq \sigma(v)_i$, transition *claim* for $\sigma(v)_i$ is not enabled.

Thus, $\mathsf{m}(\tau)(\mathit{Claimed}^{v}_{\sigma(v)_i}) = 0$. ■

We prove Theorem 3.1 for $\mathsf{templ}_{\mathsf{SPARE}}(v)$ with arbitrary claiming:

$$\forall v \in \mathsf{SPARE} : \forall \tau \in \mathsf{Traces} :$$
$$v \in \mathsf{Failed}(\tau) \iff \mathsf{m}(\tau)(\mathit{Failed}_v) > 0.$$

Proof. It suffices to show Equation (3.1) on page 62:

$$\mathsf{Eval}(\tau', v) = \mathrm{f}_v \iff \exists t_1, \ldots, t_n \in \mathsf{templ}_{\mathit{Type}(v)}(v) : \mathsf{m}(\tau') \xrightarrow{t_1} \ldots \xrightarrow{t_n} \mathsf{m}(\tau)$$
$$\wedge\ \mathsf{m}(\tau)(\mathit{Failed}_v) > 0.$$

By assumption $\mathrm{f}_v \notin \tau'$. With Definition 3.16 and induction hypothesis it holds:

$$\begin{aligned}
& \mathsf{Eval}(\tau', v) = \mathrm{f}_v \\
\iff & \mathsf{Claimed}(\tau')(v) \subseteq \mathsf{Failed}(\tau') \wedge (\sigma(v) \cap \mathsf{Available}(\tau')) = \emptyset \\
\overset{I.H.}{\iff} & \wedge\ \forall \sigma(v)_i : \left(\mathsf{m}(\tau')(\mathit{Claimed}^{v}_{\sigma(v)_i}) > 0 \implies \mathsf{m}(\tau)(\mathit{Failed}_{\sigma(v)_i}) > 0\right) \\
& \wedge \bigwedge_{1 \le i \le n} \mathsf{m}(\tau)(\mathit{Unavail}_{\sigma(v)_i}) > 0.
\end{aligned}$$

We prove both cases:

- If $\mathsf{Eval}(\tau', v) = \mathrm{f}_v$, then all claimed children have failed and the token is moved to $\mathit{Consider}_v$ by one of the transitions *child-fail*. As all places $\mathit{Unavail}_{\sigma(v)_i}$ contain a token, only transition *unavail* can fire. As a result, we have $\mathsf{m}(\tau)(\mathit{Failed}_v) > 0$.
- If $\mathsf{Eval}(\tau', v) \neq \mathrm{f}_v$, then either one claimed child is not yet failed or one child is still available.

  If not all claimed children have failed, there is a place $\mathit{Claimed}^{v}_{\sigma(v)_j}$ which contains a token, but the place $\mathit{Failed}_{\sigma(v)_j}$ does not. Then the transition *child-fail* is not enabled yet and the token cannot be moved to $\mathit{Consider}_v$. Thus, transition *unavail* cannot fire. As it is the only transition which can put a token in Failed_v, no token can be present in Failed_v.

  If all claimed children have failed, but not all places $\mathit{Unavail}_{\sigma(v)_i}$ contain a token, the 'moving' token is present in $\mathit{Consider}_v$. As at least one $\mathit{Unavail}_{\sigma(v)_j}$ contains no token, transition *unavail* is still not enabled. Thus, $\mathsf{m}(\tau)(\mathit{Failed}_v) = 0$. ■

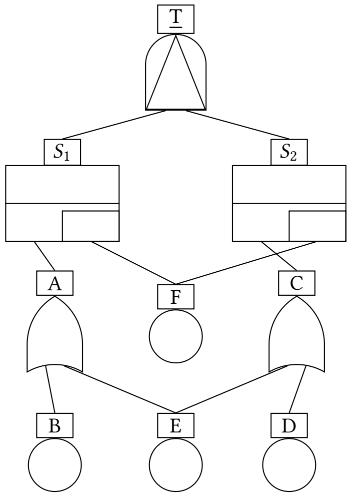


Figure 3.27: DFT containing a spare race

·spare race

Spare race | The claiming of a spare child is only possible if the newly claimed child is currently available. An operational child is available according to Definition 3.12 if it has not been claimed before by another SPARE. It is crucial to note here that it is possible for two or more SPAREs to simultaneously try to claim the same shared child. We call this a *spare race*. The question which SPARE is successful in claiming is again decided by the evaluation order. The SPARE which is evaluated first, can claim the child.

For a better understanding, we define the SPAREs which are currently claiming.

·currently claiming

Definition 3.17 (Currently claiming SPARE gates) Let $v \in \mathsf{SPARE}$, $\tau \in \mathsf{Traces}$. The *currently claiming* SPAREs after trace τ are given by:

$$v \in \mathsf{CurrentlyClaiming}(\tau) \iff \mathsf{Claimed}(\tau)(v) \subsetneq \mathsf{Failed}(\tau) \\ \wedge \sigma(v) \cap \mathsf{Available}(v) \neq \emptyset.$$

Example 3.12 (Spare race) We give the example of a spare race for the DFT depicted in Figure 3.27. Both SPAREs share the spare child F.
We consider the trace

$$\tau = \mathrm{cl}_A^{S_1}\ \mathrm{cl}_C^{S_2}\ \mathbf{f}_E\ \mathrm{f}_A\ \mathrm{f}_C.$$

After τ, SPARE S_1 has claimed the primary child A and SPARE S_2 has claimed C:

$$\mathsf{Claimed}(\tau)(S_1) = \{A\} \qquad\qquad \mathsf{Claimed}(\tau)(S_2) = \{C\}\,.$$

The failure of E leads to the failure of both A and C:

$$\mathsf{Failed}(\tau) = \{E, A, C\}\,.$$

Thus, the primary children of both S_1 and S_2 have failed simultaneously and both SPAREs need to claim a new child, i.e. $S_1, S_2 \in \mathsf{CurrentlyClaiming}(\tau)$. However, only one spare child F is available, i.e. $\mathsf{Available}(\tau) = \{F\}$. This spare child is the next child for both SPAREs:

$$\mathsf{NextSpare}(\tau, S_1) = F \qquad\qquad \mathsf{NextSpare}(\tau, S_2) = F.$$

The question which SPARE can claim F is determined by the evaluation order of the DFT semantics *SemEx*. If both $S_1, S_2 \in \mathsf{NextEval}^{SemEx}(\tau)$, then two next events are possible:

$$\mathsf{NextEvent}^{SemEx}(\tau) = \left\{\mathrm{cl}_F^{S_1}, \mathrm{cl}_F^{S_2}\right\}.$$

As a result, two substantially different traces are possible for the ftrace $\boldsymbol{\tau_\mathrm{f}} = \mathbf{f}_E\ \mathbf{f}_F$:

$$\tau_1 = \mathrm{cl}_A^{S_1}\ \mathrm{cl}_C^{S_2}\ \mathbf{f}_E\ \mathrm{f}_A\ \mathrm{f}_C\ \mathrm{cl}_F^{S_1}\ \mathrm{f}_{S_2}\ \oslash_T\ \mathbf{f}_F\ \mathrm{f}_{S_1}\ \oslash_T \in \mathsf{Traces}^{SemEx}(\boldsymbol{\tau_\mathrm{f}}).$$
$$\tau_2 = \mathrm{cl}_A^{S_1}\ \mathrm{cl}_C^{S_2}\ \mathbf{f}_E\ \mathrm{f}_A\ \mathrm{f}_C\ \mathrm{cl}_F^{S_2}\ \mathrm{f}_{S_1}\ \oslash_T\ \mathbf{f}_F\ \mathrm{f}_{S_2}\ \mathrm{f}_T \in \mathsf{Traces}^{SemEx}(\boldsymbol{\tau_\mathrm{f}}).$$

In τ_1, the TLE T is fail-safe whereas in τ_2 the PAND has failed.

As seen in the previous example, spare races can lead to quite different DFT traces and can introduce non-deterministic behaviour. In most modelling scenarios however, non-determinism is not desired and thus, spare races should be prevented.

Spare races happen when more than one SPARE is claiming at the same time, i.e. $|\mathsf{CurrentlyClaiming}(\tau)| > 1$. It is only possible to have multiple currently claiming SPAREs if children of these SPAREs fail simultaneously. Thus, one solution for spare races is to restrict the DFT structure in such a way that at most one SPARE gate can claim at a time. In other words, spare races will not occur if:

$$\forall \tau \in \mathsf{Traces} : |\mathsf{CurrentlyClaiming}(\tau)| \leq 1.$$

For *well-formed* DFTs, the following constraints are necessary to ensure that at most one SPARE is currently claiming:

- DFT Constraint 8 (Independence of modules)

- DFT Constraint 9 (No constant failed events in primary modules)
- DFT Constraint 10 (No sharing of primary modules)

The first constraint is already assumed throughout this chapter. The other two constraints ensure that the initial claiming of the primary modules is deterministic since each SPARE gets to claim its first child. Thus, no spare races can occur in the beginning. Junges proved in his master's thesis [Jun15, Theorem 4.30] that for all DFTs satisfying these three constraints at most one SPARE is claiming at all times.

A second solution for spare races is to explicitly forbid that multiple SPAREs are currently claiming. This can be ensured by defining the evaluation order such that at most one SPARE is evaluated next. In other words, it must hold:

$$\forall \tau \in \mathsf{Traces}:$$
$$|\mathsf{NextEval}^{Sem}(\tau) \cap \mathsf{SPARE}_{\mathcal{F}}| \leq 1.$$

We revisit the spare race of Example 3.12 and discuss how spare races are handled in both solution variants. The first solution of restricting the DFT renders the DFT in Figure 3.27 invalid. The spare modules of A and B are not independent, because they both contain E. Thus, the DFT needs to be adapted.

The second solution requires a modified evaluation order. For example, one could define $SemEx'$ such that in case of possible spare races, SPARE S_1 is always evaluated before S_2. That means $\mathsf{NextEvent}^{SemEx'}(\tau) = \left\{\mathrm{cl}_F^{S_1}\right\}$ and only trace $\tau_1 \in \mathsf{Traces}^{SemEx'}(\boldsymbol{\tau}_\mathbf{f})$ is valid any more. Trace $\tau_2 \notin \mathsf{Traces}^{SemEx'}(\boldsymbol{\tau}_\mathbf{f})$ cannot happen any more.

3.6.4 Activation

·activation

Apart from claiming, the SPARE gate has a second unique concept: *activation*. Whenever a spare child is claimed, all elements belonging to the same spare module are activated. Activation changes the failure behaviour of SEs. Active SEs fail according to their *active* failure distribution FD_a. Similarly, passive SEs which are not in use fail according to their *passive* failure distribution FD_p.

·cold

·hot

·warm

Originally, the type of the SPARE gate defined the passive failure rates of its children [DVG97]. Three types of SPARE gate were introduced: cold SPARE, hot SPARE and warm SPARE. In the *cold SPARE* (also called csp) only the claimed children can fail. All unused children are *cold SEs* and cannot fail. They are modelled by having a dormancy factor of $d = 0$. In the *hot SPARE* (hsp) by contrast it makes no difference for the failure rate whether the child is active or passive. In other words $\lambda_p = \lambda_a$ which is modelled by dormancy factor $d = 1$. While cold and hot SPARE represent the extremes of the spectrum, the *warm SPARE* (wsp) represents the middle. A child of a warm SPARE can fail in passive mode, but has a lower failure rate compared to active use, i.e. $0 < \lambda_p < \lambda_a$. This is modelled by a dormancy factor $0 < d < 1$.

As seen, all three types of SPARE gates can be modelled by setting the corresponding dormancy factor. The logical step therefore was to move the passive failure behaviour from the type of SPARE gate to the SE itself [BCS10]. Instead of having three types of SPARE gates, we have one general SPARE gate and use the passive failure distribution of each SE to specify the behaviour while not in use. By letting the SE define which passive failure rate to use, also semantic issues regarding the passive failure distribution of shared spares are avoided. In the old setting, it was not clear what happens if shared spares are for example children of both cold and hot SPAREs (cf. [CSD00]).

Activation is performed per (spare) *module*. If a new child is claimed, it gets activated. Not only the child but the complete spare module represented by the spare child is activated. A DFT element v is *activated* after τ iff either it is part of the *top module* or its module is claimed by an active SPARE. We define the activation behaviour for all elements.

activated

Definition 3.18 (Activation) Let $\mathcal{F} = (V, \sigma, \mathit{Type}, \mathit{top}, \Theta)$ be a DFT, $\tau \in \mathsf{Traces}$. The activation condition for $v \in \mathcal{F}$ is defined as follows:

- $v = \mathit{top}$: The condition $\mathsf{Act}(\tau, v)$ always holds.
- $v \in \mathsf{SpareModuleRepr}$:

$$\mathsf{Act}(\tau, v) \iff \exists v' \in \rho(v) : v' \in \mathsf{SPARE} \wedge v' \in \mathsf{Active}(\tau) \wedge v \in \mathsf{Claimed}(\tau)(v').$$

- $v \notin \mathsf{ModuleRepr}$:

$$\mathsf{Act}(\tau, v) \iff \exists v' \in \mathsf{ModuleRepr} : v \in \mathsf{Module}(v') \wedge v' \in \mathsf{Active}(\tau).$$

GSPN template | Activation is integrated in the GSPN framework in two ways: we extend the existing template for SPAREs to also integrate activation and we add separate activation templates for each module representative. The activation templates use the interface place Active_v to store the information whether an element v is activated.

We give the template extension for activation in a SPARE gate in Figure 3.28. The GSPN extension contains the interface places $\mathit{Active}_v, \mathit{Active}_{\sigma(v)_1}, \ldots, \mathit{Active}_{\sigma(v)_n}$. Note that the places $\mathit{Claimed}^v_{\sigma(v)_1}, \ldots, \mathit{Claimed}^v_{\sigma(v)_n}$ represent the places from the original SPARE template in Figure 3.26 on page 98 and connect the activation extension to the original template. Whenever SPARE v is active and a child $\sigma(v)_i$ is claimed, places Active_v and $\mathit{Claimed}^v_{\sigma(v)_i}$ contain a token. The corresponding immediate transition can fire and places a token in $\mathit{Active}_{\sigma(v)_i}$ indicating that the child $\sigma(v)_i$ is activated now.

The top module is activated by placing a token in $\mathit{Active}_{\mathit{top}}$. This is handled by the *initialisation template* $\mathsf{templ}_{\mathsf{init}}$ which will be introduced later in Figure 3.42 on page 125.

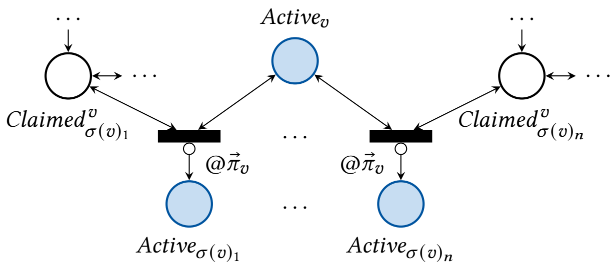


Figure 3.28: GSPN template for activation in SPARE

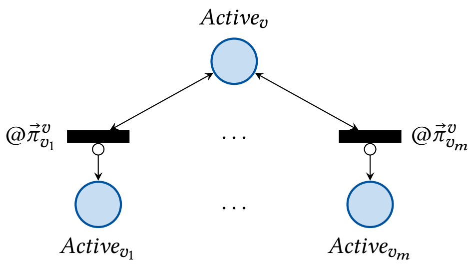


Figure 3.29: GSPN template for activation in modules

We have seen how to activate the module representatives. It is however required to activate all elements in a module if the *module representative* has been activated. This is handled by the activation template $\mathrm{templ}_{\mathrm{act}}(v)$ for each $v \in \mathrm{ModuleRepr}$ as presented in Figure 3.29. Let $\{v_1, \ldots, v_m, v\} = \mathrm{Module}(v)$ be the members of the module for v. Whenever the place $Active_v$ of the module representative v contains a token, all members $v_1, \ldots, v_m$ of the corresponding module are activated through the intermediate transitions. We use different priority variables $\vec{\pi}^v_{v_1}, \ldots, \vec{\pi}^v_{v_m}$ to accurately reflect the evaluation order for each element v_i.

The templates for activation presented here significantly differ from the original ones in [4]. The template extension for SPAREs is the same as before. However, in [4] there is an activation template for each gate which propagates the activation from the gate to its children. This approach however does neglect some special cases. We depict them by the example DFT in Figure 3.30. All elements are part of the top module and therefore should be activated once the TLE T is activated. However, the original activation template [4, Fig. 8(a)] only allows 'downwards' propagation of activation. That means only A will be activated from T. B will not be activated because FDEPs do not propagate activation and D cannot be activated, because it is a parent of A

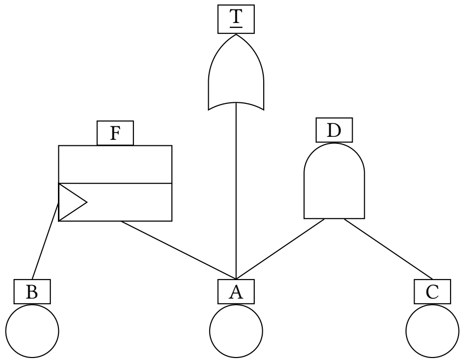


Figure 3.30: Example DFT depicting special cases for activation

and not a child. The newly introduced activation template for module representatives in Figure 3.29 avoids these issues by directly activating all elements in a module. Instead of propagating the activation through all gates, each member of a module is directly activated from the module representative.

We conclude this section by showing the correctness of the activation templates, i.e. proving Theorem 3.2:

$$\forall v \in \mathcal{F} : \forall \tau \in \mathsf{Traces} :$$
$$v \in \mathsf{Active}(\tau) \iff \mathsf{m}(\tau)(\mathit{Active}_v) > 0.$$

Proof. We perform an induction over τ similar to the proof for Theorem 3.1:

- $\tau = \varepsilon$: We know $\mathsf{Active}(\varepsilon) = \emptyset$ and by construction $\mathsf{m}(\varepsilon)(\mathit{Active}_v) = 0 \ \forall v \in \mathcal{F}$.
- $\tau = \tau'\ e$: We assume $e = \uparrow_v$. Otherwise the statement holds by induction hypothesis. It remains to show:

$$\mathsf{Act}(\tau', v) \iff \mathsf{m}(\tau)(\mathit{Active}_v) > 0.$$

We distinguish the different DFT types:

- $v = \mathit{top}$: Then $\mathsf{Act}(\tau, v)$ always holds. Using the initialisation template $\mathsf{templ}_{\mathsf{init}}$ from Section 3.9.1, we have $\mathsf{m}(\tau)(\mathit{Active}_{top}) > 0$ for all traces $\tau \neq \varepsilon$.

– $v \in \mathsf{SpareModuleRepr}$: Then

$$\begin{aligned}\mathsf{Act}(\tau', v) &\iff \exists v' \in \rho(v) : v' \in \mathsf{SPARE}\\ &\qquad \wedge v' \in \mathsf{Active}(\tau') \wedge v \in \mathsf{Claimed}(\tau')(v')\\ &\overset{I.H.}{\iff} \exists v' \in \rho(v) : v' \in \mathsf{SPARE}\\ &\qquad \wedge \mathsf{m}(\tau')(\mathit{Active}_{v'}) > 0 \wedge \mathsf{m}(\tau')(\mathit{Claimed}_v^{v'}) > 0.\end{aligned}$$

As the parent is a SPARE, we consider the template extension for SPAREs. We know that both $\mathit{Claimed}_v^{v'}$ and $\mathit{Active}_{v'}$ contain a token. Thus, the corresponding immediate transition can fire and places a token in Active_v.

– $v \notin \mathsf{ModuleRepr}$: Then

$$\begin{aligned}\mathsf{Act}(\tau', v) &\iff \exists v' \in \mathsf{ModuleRepr} : v \in \mathsf{Module}(v') \wedge v' \in \mathsf{Active}(\tau')\\ &\overset{I.H.}{\iff} \exists v' \in \mathsf{ModuleRepr} : v \in \mathsf{Module}(v') \wedge \mathsf{m}(\tau')(\mathit{Active}_{v'}) > 0.\end{aligned}$$

We consider $\mathsf{templ}_{\mathsf{act}}(v')$ and know that $\mathit{Active}_{v'}$ contains a token. Thus, all immediate transitions are in concession. As $v \in \mathsf{Module}(v')$ we have $v = v_i$ for one $1 \leq i \leq m$. W.l.o.g. we assume $v = v_1$. Firing the corresponding transition places a token in Active_{v_1}. Thus, $\mathsf{m}(\tau)(\mathit{Active}_v) > 0$. ■

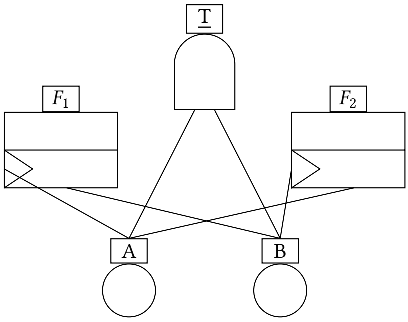
3.7 Dependencies

In complex models it is often the case that failures of specific components lead to failures of other components. In other words, the functionality of components depends on the correct operation of other components, they are functionally dependent. An example of functional dependencies is a power supply which powers several servers. Each server can fail on its own due to malfunctioning hardware components. It is also possible, that the power supply fails. In that case, all servers would immediately fail—if no backup power supply is available. The servers are therefore functionally dependent on the power supply.

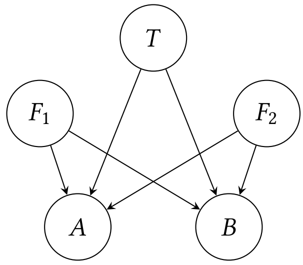
dependency ·trigger ·dependent elements ·forwarding

In DFTs we use *dependencies* to model such failure connections between elements. The first child of a dependency is called the *trigger* and the other children are called *dependent elements*. The failure of the trigger leads to the failure of the dependent elements. That means the dependent elements fail due to the outside failure of the trigger and not because of any internal failure of the component itself. We also say the failure is *forwarded* to the dependent elements.

The dependency itself only represents a causal connection between components.



(a) DFT $\mathcal{F}$



(b) Graph $G(\mathcal{F})$

Figure 3.31: Feedback loop with dependencies

Therefore, the dependency itself cannot fail and is not the child of any other element (cf. DFT Constraint 5 on page 26 (No parents for SEQ and PDEP)).

Moreover, dependencies allow to model *feedback loops* between components as shown in Figure 3.31a. The failure of A is forwarded by F_1 and lets B fail as well. Similarly, the failure of B leads to a failure of A through F_2. We call this behaviour cyclic dependency.

Definition 3.19 (Cyclic dependencies) Let $\mathcal{F}$ be a DFT and $F \subseteq \mathrm{PDEP}_{\mathcal{F}}$ be a set of dependencies. The dependency edges for F are added to the graph of $\mathcal{F}$ (cf. Definition 2.12 on page 23) by inserting edges from the dependent elements of a dependency to the trigger:

$$E^{dep}(F) = \{(\sigma(f)_i, \sigma(f)_1) \mid f \in F, 2 \leq i \leq |\sigma(f)|\}\,.$$

We call dependencies $F \subseteq \mathrm{PDEP}_{\mathcal{F}}$ *cyclic* if the graph of $\mathcal{F}$ extended with the dependency edges of F, i.e. $\big(V, E(\sigma) \cup E^{dep}(F)\big)$, contains a cycle. Additionally, F needs to be minimal, i.e. all proper subsets $F' \subset F$ are not cyclic dependencies. ·cyclic

Note that cyclic dependencies still yield an acyclic *graph*. In the example, A and B are cyclic dependencies. However, the graph $G(\mathcal{F})$ (depicted in Figure 3.31b) is acyclic, because A and B are both children of the FDEPs F_1 and F_2.

Also note that single dependencies can already form a cyclic dependency if they forward the failure *downwards*. An example DFT is given in Figure 4.19 on page 171.

Gates as dependent children | We restrict the dependent children to BEs (cf. DFT Constraint 11 (Dependent events of PDEPs are BEs)). This restriction could be lifted to arbitrary gates. However, the behaviour of triggered SPARE gates for example needs to

(a) FDEP (b) Alternative representation of FDEP

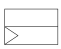
Figure 3.32: Graphical representation of FDEP

be specified in these cases. If a SPARE fails due to a trigger, one of the claimed children is still operational. It might make sense to release this child again, i.e. to 'unclaim' it, such that another SPARE gate might use it. This however requires the introduction of new DFT events and might introduce further semantic questions. Other works such as [BCS10] do not consider a triggered SPARE as failed. As a result, the SPARE can still claim and activate children even though it is already failed. We agree with Junges et al. that 'FDEPs with dependent gates require special treatment' [JGKS16] and therefore restrict ourselves to BEs as dependent children. This restriction is also consistent with other works, cf. e.g. [DBB92; RS15]. The trigger however can be an arbitrary gate.

·probabilistic dependency

·functional dependency

We distinguish between two types of dependencies. The *probabilistic dependency* PDEP_p only forwards the failure with a certain probability p. If the trigger fails, the failure is forwarded to the dependent children with probability p. With probability $1-p$ no forwarding takes place and the dependent children do not fail. The *functional dependency* FDEP always forwards the failure. The functional dependency can be expressed by a PDEP_1.

3.7.1  Functional dependency (FDEP)

We start by introducing the more commonly used FDEP [DBB90]. The graphical representation of an FDEP is given in Figure 3.32. The FDEP gate is depicted in Figure 3.32a. The trigger A is connected to the input triangle of the FDEP, the dependent child B is given as usual at the bottom of F. As FDEPs commonly connect parts of the DFT which might be far apart, the DFT visualisation can quickly become cluttered and hard to follow. We therefore often employ a simpler representation for binary FDEPs by using double dashed arrows as shown in Figure 3.32b. The source of the arrow represents the trigger and the target denotes the dependent child.

We now define the behaviour of the functional dependency. For simplicity we only consider a binary FDEP with exactly one dependent child.

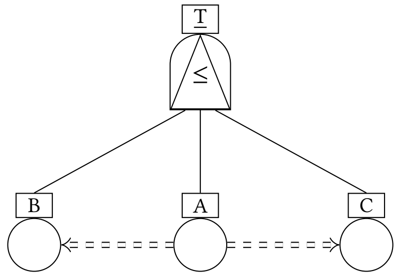


Figure 3.33: Forwarding of failures in FDEPs

Definition 3.20 (Evaluation of FDEP) Let $v \in \mathrm{FDEP}_{\mathcal{F}}$ and $\tau \in \mathrm{Traces}$. The evaluation function is given as follows:

$$\mathrm{Eval}(\tau, v) = \begin{cases} \mathrm{f}_{\sigma(v)_2} & \text{if } \mathrm{f}_{\sigma(v)_2} \notin \tau \wedge \mathrm{f}_{\sigma(v)_1} \in \tau \wedge \mathrm{f}_{\sigma(v)_2} \notin \mathrm{Disabled}(\tau) \\ \oslash_v & \text{otherwise.} \end{cases}$$

Upon failure of the first child, the other child fails as well. However, we also have to consider a special case which can be introduced by restrictors. If the dependent child is *disabled* by a restrictor, then it cannot fail. Failure of disabled elements would lead to invalid traces and is therefore prevented. However, if later on the child becomes enabled, it will immediately fail due to the dependency. We will consider the details of this behaviour later in Section 4.2.2 on page 150.

Note that if we had multiple dependent children, it would not be clear which child should fail first. We therefore restrict FDEPs to the binary case. An FDEP with multiple dependent children can be translated into multiple binary FDEPs while still capturing the desired behaviour. The order in which the children should fail can then be specified by the order in which each FDEP is evaluated.

Example 3.13 (Failure forwarding in FDEPs) Consider the DFT given in Figure 3.33 and the ftrace $\boldsymbol{\tau_{\mathrm{f}}} = \mathbf{f}_A$. There are different possibilities how to forward the failure of A via the FDEPs. We give some possible traces and the corresponding observed trace order for T:

$$\begin{aligned} \tau_1 &= \mathbf{f}_A\, \mathrm{f}_B\, \mathrm{f}_C\, \mathrm{f}_T && \text{no order observed} \\ \tau_2 &= \mathbf{f}_A\, \oslash_T\, \mathrm{f}_B\, \mathrm{f}_C\, \oslash_T && \mathbf{f}_A \blacktriangleleft_T^{\tau_2} \mathrm{f}_B, \mathbf{f}_A \blacktriangleleft_T^{\tau_2} \mathrm{f}_C \\ \tau_3 &= \mathbf{f}_A\, \oslash_T\, \mathrm{f}_B\, \oslash_T\, \mathrm{f}_C\, \oslash_T && \mathbf{f}_A \blacktriangleleft_T^{\tau_3} \mathrm{f}_B \blacktriangleleft_T^{\tau_3} \mathrm{f}_C \\ \tau_4 &= \mathbf{f}_A\, \oslash_T\, \mathrm{f}_C\, \oslash_T\, \mathrm{f}_B\, \oslash_T && \mathbf{f}_A \blacktriangleleft_T^{\tau_4} \mathrm{f}_C \blacktriangleleft_T^{\tau_4} \mathrm{f}_B \end{aligned}$$

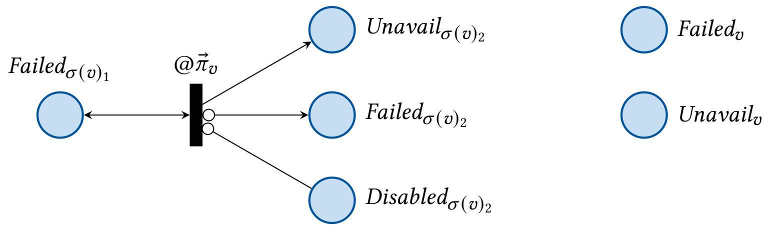


Figure 3.34: GSPN template for FDEP

Trace τ_1 lets the forwarding happen simultaneously to the failure of A and no order can be observed. The trace τ_1 therefore does not capture the causal relationship between e.g. the failures of A and B. Just by looking at the observed trace it is not clear, that only A failed on its own, but B and C were triggered by A.
Trace τ_2 incorporates the causal relationship. From this trace it is clear, that the failures of B and C causally depend on the failure of A and therefore take place after the failure of A. Notice that PAND T does not fail in τ_2, because the children did not fail in the correct observed order.
In some instances it might even be important to have a total ordering on the failure of BEs. One reason would be that for a total ordering spare races are not possible any more, because then only one SPARE can fail at a time. Traces τ_3 and τ_4 represent a total order.

In summary, we see that the ordering of FDEP forwarding is not trivial and can cause significant differences in the failure behaviour of the complete DFT. We will investigate these issues in detail in Section 4.4.2 on page 163.

GSPN template | The GSPN template for an FDEP is given in Figure 3.34. If the trigger has failed, i.e. a token is present in $Failed_{\sigma(v)_1}$, the failure can be forwarded to the child via the immediate transition. The dependent child $\sigma(v)_2$ can fail if it is not yet failed and its failure is not disabled. This is ensured by the two inhibitor arcs from $Failed_{\sigma(v)_2}$ and $Disabled_{\sigma(v)_2}$. If all preconditions are satisfied, the immediate transition fires and places tokens in $Failed_{\sigma(v)_2}$ and $Unavail_{\sigma(v)_2}$. Note that the interface places $Failed_v$ and $Unavail_v$ of the FDEP v are isolated because the FDEP itself cannot fail.

Correctness of failure behaviour | We prove the correctness of $\mathsf{templ}_{\mathsf{FDEP}}(v)$:

$$\forall v \in \mathsf{FDEP} : \forall \tau \in \mathsf{Traces} :$$
$$\sigma(v)_2 \in \mathsf{Failed}(\tau) \iff \mathsf{m}(\tau)(Failed_{\sigma(v)_2}) > 0.$$

Proof. Note that not the failure of the FDEP v itself but of the second child is considered here. We therefore also have to consider the case where v is not evaluated, but $\sigma(v)_2$ still fails, because the BE fails on its own. We already considered this case before in the correctness proof for SE_{exp} and showed

$$\sigma(v)_2 \in \mathsf{Failed}(\tau) \iff \mathsf{m}(\tau)(\mathit{Failed}_{\sigma(v)_2}) > 0.$$

It therefore remains to show a slight variation of Equation (3.1) on page 62:

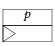
$$\begin{aligned}\mathsf{Eval}(\tau', v) = \mathsf{f}_{\sigma(v)_2} \iff \exists t_1, \ldots, t_n \in \mathsf{templ}_{\mathsf{FDEP}}(v) : \mathsf{m}(\tau') \xrightarrow{t_1} \ldots \xrightarrow{t_n} \mathsf{m}(\tau) \\ \wedge\, \mathsf{m}(\tau)(\mathit{Failed}_{\sigma(v)_2}) > 0.\end{aligned}$$

We have

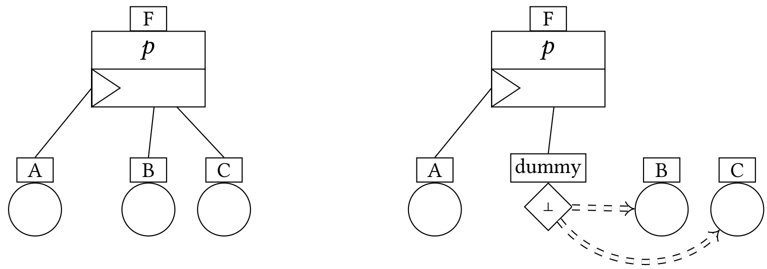
$$\begin{aligned} & \mathsf{Eval}(\tau', v) = \mathsf{f}_{\sigma(v)_2} \\ \iff & \mathsf{f}_{\sigma(v)_2} \notin \tau' \wedge \mathsf{f}_{\sigma(v)_1} \in \tau' \wedge \mathsf{f}_{\sigma(v)_2} \notin \mathsf{Disabled}(\tau') \\ \overset{I.H.}{\iff} & \mathsf{m}(\tau')(\mathit{Failed}_{\sigma(v)_2}) = 0 \wedge \mathsf{m}(\tau')(\mathit{Failed}_{\sigma(v)_1}) > 0 \wedge \mathsf{m}(\tau')(\mathit{Disabled}_{\sigma(v)_2}) = 0.\end{aligned}$$

We start with the case $\mathsf{Eval}(\tau', v) = \mathsf{f}_{\sigma(v)_2}$. That means $\sigma(v)_2 \in \mathsf{Failed}(\tau)$. Furthermore, in $\mathsf{templ}_{\mathsf{FDEP}}(v)$ the immediate transition is enabled, because neither $\mathit{Failed}_{\sigma(v)_2}$ nor $\mathit{Disabled}_{\sigma(v)_2}$ contain a token and $\mathit{Failed}_{\sigma(v)_1}$ contains a token. Firing the transition results in a token in $\mathit{Failed}_{\sigma(v)_2}$ and the statement holds.
We consider the case $\mathsf{Eval}(\tau', v) = \oslash_v$. That means $\sigma(v)_2 \notin \mathsf{Failed}(\tau)$. By assumption, $\sigma(v)_2 \notin \mathsf{Failed}(\tau')$ and therefore $\mathsf{m}(\tau')(\mathit{Failed}_{\sigma(v)_2}) = 0$. In the GSPN template, the immediate transition is not enabled, because either $\mathit{Failed}_{\sigma(v)_1}$ does not contain a token or $\mathit{Disabled}_{\sigma(v)_2}$ does contain a token. That means the immediate transition cannot fire in $\mathsf{m}(\tau')$ and thus,

$$\mathsf{m}(\tau)(\mathit{Failed}_{\sigma(v)_2}) = \mathsf{m}(\tau')(\mathit{Failed}_{\sigma(v)_2}) = 0. \qquad \blacksquare$$

3.7.2  Probabilistic dependency (PDEP)

We consider the more general PDEP [MPB05] now. The PDEP_p employs a two step process. First, a coin flip with bias p is performed. With probability p the failure will be forwarded and we continue with the second step in which all dependent children fail similar to the FDEP. With probability $1 - p$ the failure will not be forwarded and the process stops. In this case, all dependent children keep working and do not fail. Note that the coin flip is performed once for the complete PDEP and not individually for each dependent child.



(a) PDEP with multiple dependent children (b) Binary PDEP

Figure 3.35: Example translation for binary PDEP

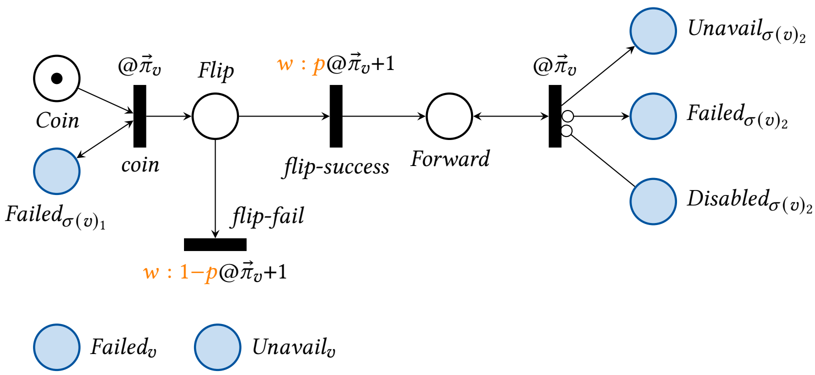


Figure 3.36: GSPN template for PDEP

For simplicity, we again consider only binary PDEPs. We can translate each PDEP with multiple dependent children into a binary PDEP and FDEPs as shown in Figure 3.35.

GSPN template | We present the GSPN template for a binary PDEP_p in Figure 3.36. The template has three auxiliary places *Coin*, *Flip* and *Forward*. Initially, a token is present in *Coin*. If the trigger fails, i.e. a token is present in $Failed_{\sigma(v)_1}$, the immediate transition *coin* is enabled. Firing *coin* moves the token from *Coin* to *Flip* which ensures that the coin flip can only be performed once. If a token is in *Flip*, both transitions *flip-success* and *flip-fail* are in concession. Both immediate transitions have a unique priority $\vec{\pi}_v + 1$ which is not occurring elsewhere and are part of the same *partitioning*. As a result, both transitions will be enabled at the same time and the *conflict* will be resolved by random choice. The *weights* $w = p$ (for *flip-success*) and $w = 1 - p$ (for *flip-fail*) ensure that each transition fires with the correct probability. Firing transition *flip-fail* removes the token from *Flip* and no further action is possible. Firing transition *flip-success* moves the token to place *Forward*. The remaining (right) part of the GSPN template equates to the GSPN template for the FDEP. Placing a token in *Forward* therefore ensures that the failure is forwarded to all dependent children.

We can easily see that PDEP_1 behaves as FDEP, because in this instance transition *flip-fail* can never fire and the token from $\mathit{Failed}_{\sigma(v)_1}$ is always duplicated in *Forward*.

3.8 Restrictors

The last type of element in DFTs are *restrictors* which forbid certain sequences of failures. We indicate sequences which violate the ordering requirement by event $\lightning$. A restrictor disables and enables the occurrence of certain failures. Disabled elements are not allowed to fail, because their failure would lead to invalid traces. Disabled elements might be enabled after the failure of other components and can fail then. restrictor

3.8.1 $\boxed{\rightarrow}$ Sequence enforcer (SEQ)

The most commonly used restrictor is the sequence enforcer (SEQ) [DBB92]. A SEQ ensures that failures of its children can only occur in order from left to right. This ordering requirement is very similar to the failure condition of a PAND. However, their behaviour is quite different. In a SEQ, children cannot fail out of order. In a PAND in contrast, children can fail out of order, but the PAND becomes *fail-safe*.

Analogously to PAND we have the question of simultaneous failures for SEQ. We therefore again distinguish between inclusive and exclusive variants of SEQ. In an *inclusive* $\mathsf{SEQ}_{\leq}$, failures occurring simultaneously are considered as valid. In an *exclusive* $\mathsf{SEQ}_{<}$, all failures have to be strictly ordered and simultaneous failures are not allowed. ·inclusive ·exclusive

Definition 3.21 (Evaluation of SEQ) Let $\mathcal{F} = (V, \sigma, \mathit{Type}, \mathit{top}, \Theta)$ be a DFT and $\tau \in \mathsf{Traces}$. The evaluation function for $v \in \mathsf{SEQ}_{\mathcal{F}}$ is given as follows:

$$\mathsf{Eval}(\tau, v) = \begin{cases} \lightning_v & \text{if } \lightning_v \notin \tau \wedge \mathsf{Viol}(\tau, v) \\ \oslash_v & \text{otherwise,} \end{cases}$$

where the violation condition is defined as follows:

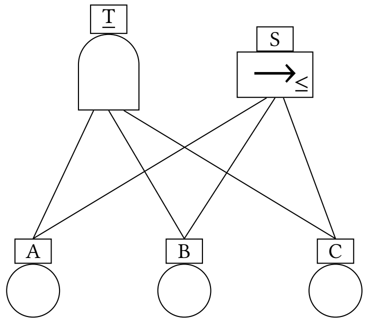


Figure 3.37: Example DFT with $\mathrm{SEQ}_{\leq}$

- For $v \in \mathrm{SEQ}_{\leq}$:

$$\begin{aligned}\mathsf{Viol}(\tau, v) &\iff \neg \bigwedge_{i=2}^{n} \left(\mathrm{f}_{\sigma(v)_i} \in \tau \implies \left(\mathrm{f}_{\sigma(v)_{i-1}} \in \tau \wedge \mathrm{f}_{\sigma(v)_i} \not\blacktriangleleft^{\tau}_{v} \mathrm{f}_{\sigma(v)_{i-1}}\right)\right) \\ &\iff \bigvee_{i=2}^{n} \left(\mathrm{f}_{\sigma(v)_i} \blacktriangleleft^{\tau}_{v} \mathrm{f}_{\sigma(v)_{i-1}} \vee \left(\mathrm{f}_{\sigma(v)_{i-1}} \notin \tau \wedge \mathrm{f}_{\sigma(v)_i} \in \tau\right)\right)\end{aligned}$$

- For $v \in \mathrm{SEQ}_{<}$:

$$\begin{aligned}\mathsf{Viol}(\tau, v) &\iff \neg \bigwedge_{i=2}^{n} \left(\mathrm{f}_{\sigma(v)_i} \in \tau \implies \left(\mathrm{f}_{\sigma(v)_{i-1}} \blacktriangleleft^{\tau}_{v} \mathrm{f}_{\sigma(v)_i}\right)\right) \\ &\iff \bigvee_{i=1}^{n-1} \left(\left(\mathrm{f}_{\sigma(v)_i} \in \tau \wedge \mathrm{f}_{\sigma(v)_{i-1}} \not\blacktriangleleft^{\tau}_{v} \mathrm{f}_{\sigma(v)_i}\right) \vee \left(\mathrm{f}_{\sigma(v)_{i-1}} \notin \tau \wedge \mathrm{f}_{\sigma(v)_i} \in \tau\right)\right)\end{aligned}$$

The order of a $\mathrm{SEQ}_{\leq}$ is violated if a child fails before its left sibling. That means either the left sibling fails strictly after the child or not at all. Similarly, the order of a $\mathrm{SEQ}_{<}$ is violated if the child fails and either the left sibling does not fail at all or at least it does not fail before the child.

SEQs limit the valid traces in a DFT because they remove certain orders of failures.

Example 3.14 (Sequence enforcer) We depict a DFT containing a $\mathrm{SEQ}_{\leq}$ in Figure 3.37. Two possible traces are:

$$\begin{aligned}\tau_1 &= \mathrm{f}_A \oslash_S \mathrm{f}_B \oslash_S \mathrm{f}_C \oslash_S \mathrm{f}_T \\ \tau_2 &= \mathrm{f}_B \lightning_S \mathrm{f}_A\ \mathrm{f}_C\ \mathrm{f}_T\end{aligned}$$

Only the first trace is valid, i.e. $\tau_1 \in \mathsf{Traces}(\mathrm{f}_A\ \mathrm{f}_B\ \mathrm{f}_C)$. Trace $\tau_2 \notin \mathsf{Traces}(\mathrm{f}_B\ \mathrm{f}_A\ \mathrm{f}_C)$ contains $\lightning_S$ and is therefore invalid.
We look at τ_1 in detail. In the beginning, only A can fail because it is the leftmost child. Child B and C are disabled and cannot fail, because A has not failed yet. After the failure of A, child B becomes enabled and can fail. C is still disabled. Only after the failure of B is C enabled and can fail.

We have given the general definition for disabled elements in Definition 2.34 on page 41. Due to DFT Constraint 13 (Inputs of SEQs are SEs) and DFT Constraint 14 (No dependencies to inputs of SEQ), we can give a more concrete definition for the disabled elements of SEQs. If there is a child left of $\sigma(v)_i$ which is not yet failed, child $\sigma(v)_i$ is disabled. A failure of $\sigma(v)_i$ can only occur if it is not disabled, i.e. the children $\sigma(v)_1, \ldots, \sigma(v)_{i-1}$ have already failed.

Lemma 3.13 (Disabled elements for SEQ) Let $\mathcal{F} = (V, \sigma, \mathit{Type}, \mathit{top}, \Theta)$ be a DFT and let $\tau \in \mathsf{Traces}$.
The *disabled elements* for $\mathrm{SEQ}_\leq$ and $\mathrm{SEQ}_<$ are given as: ·SEQ

$$v \in \mathsf{Disabled}(\tau) \iff v \notin \mathsf{Failed}(\tau) \land \exists s \in \mathrm{SEQ}_\mathcal{F} : \exists 1 \leq j < i \leq |\sigma(s)| : \\ v = \sigma(s)_i \land \sigma(s)_j \notin \mathsf{Failed}(\tau).$$

Proof. We start with the direction '$\Longleftarrow$'. The only interesting case is the trace $\tau\ \mathrm{f}_v$. By assumption, there is a child $\sigma(s)_j$ left of v which is not failed in τ and therefore $\mathrm{f}_{\sigma(s)_j} \notin \tau\ \mathrm{f}_v$. We also know $\mathrm{f}_v = \mathrm{f}_{\sigma(s)_i} \in \tau\ \mathrm{f}_v$. That means there exists $1 \leq k < i$ such that

$$\mathrm{f}_{\sigma(s)_k} \notin \tau\ \mathrm{f}_v \land \mathrm{f}_{\sigma(s)_{k+1}} \in \tau\ \mathrm{f}_v.$$

We consider the propagation step after $\tau\ \mathrm{f}_v$. Let τ' be the trace after which SEQ s is evaluated next, i.e. $s \in \mathsf{NextEval}^{\mathit{Sem}}(\tau')$ and $\tau\ \mathrm{f}_v \subseteq \tau'$. Due to DFT Constraint 13 (Inputs of SEQs are SEs) and DFT Constraint 14 (No dependencies to inputs of SEQ), no child of s can fail in-between $\tau\ \mathrm{f}_v$ and τ'. We therefore know

$$\mathrm{f}_{\sigma(s)_k} \notin \tau \implies \mathrm{f}_{\sigma(s)_k} \notin \tau'$$

and it holds

$$\mathrm{f}_{\sigma(s)_k} \notin \tau' \land \mathrm{f}_{\sigma(s)_{k+1}} \in \tau'.$$

Thus, the second part of $\mathsf{Viol}(\tau', s)$ holds—for both inclusive and exclusive SEQ—and therefore $\mathsf{Eval}(\tau', s) = \lightning_s$. In summary, we have

$$\mathsf{Propagate}^{\mathit{Sem}}(\tau\ \mathrm{f}_v) \subseteq \lightning^{\blacktriangleright}.$$

and by Definition 2.34 it holds $v \in \mathsf{Disabled}(\tau)$.
We show the direction '$\implies$' by contraposition. We assume

$$v \in \mathsf{Failed}(\tau) \vee \forall s \in \mathrm{SEQ}_{\mathcal{F}} : \forall 1 \leq j < i \leq |\sigma(s)| : v = \sigma(s)_i \implies \sigma(s)_j \in \mathsf{Failed}(\tau).$$

If $v \in \mathsf{Failed}(\tau)$, then by Definition 2.34 it directly follows $v \notin \mathsf{Disabled}(v)$. For the remainder we assume $v \notin \mathsf{Failed}(\tau)$. We consider the failure of v and $\mathsf{Propagate}^{Sem}(\tau\, \mathsf{f}_v)$. Let τ' again be the trace after which SEQ s is evaluated. As $\tau \in \mathsf{Traces}$ is a valid trace, $\mathsf{Viol}(\tau, s)$ did not hold before and in particular it holds

$$\bigwedge_{k=2}^{n} \left(\mathsf{f}_{\sigma(s)_k} \in \tau \implies \left(\mathsf{f}_{\sigma(s)_{k-1}} \blacktriangleleft_s^{\tau} \mathsf{f}_{\sigma(s)_k}\right)\right).$$

We know that all children left of v were already failed in τ. Thus, $\mathsf{f}_{\sigma(s)_j} \blacktriangleleft_s^{\tau'} \mathsf{f}_{\sigma(s)_i}$ for all $j < i$ and in particular:

$$\mathsf{f}_{\sigma(s)_i} \in \tau' \wedge \mathsf{f}_{\sigma(s)_{i-1}} \blacktriangleleft_s^{\tau'} \mathsf{f}_{\sigma(s)_i}.$$

Again, because of DFT Constraint 13 (Inputs of SEQs are SEs) and DFT Constraint 14 (No dependencies to inputs of SEQ) no child of s fails in-between $\tau\, \mathsf{f}_v$ and τ'. That means, $\mathsf{f}_{\sigma(s)_k} \notin \tau'$ for all $k > i$. In summary, $\mathsf{Viol}(\tau', s)$ does not hold, $\mathsf{Eval}(\tau', s) = \oslash_s$ and thus, $v \notin \mathsf{Disabled}(\tau)$. ■

We can see that the variant of SEQ—inclusive or exclusive—is not important for the calculation of the disabled elements in Lemma 3.13.

Corollary 3.14 ($\mathrm{SEQ}_{\leq}$ and $\mathrm{SEQ}_{<}$ coincide under certain constraints) If both DFT Constraint 13 (Inputs of SEQs are SEs) and DFT Constraint 14 (No dependencies to inputs of SEQ) hold, then both variants $\mathrm{SEQ}_{\leq}$ and $\mathrm{SEQ}_{<}$ coincide.

In the following, we do not distinguish between $\mathrm{SEQ}_{\leq}$ and $\mathrm{SEQ}_{<}$ any more, because they coincide for the given DFT restrictions.

GSPN template | We present the GSPN template for a SEQ with n children in Figure 3.38. The template makes use of the interface place $Disabled_v$ for each BE v. Tokens in $Disabled_v$ indicate that v is disabled and can currently not fail.

The tokens in $Disabled_v$ are removed if v becomes enabled. Thus, the places $Disabled_v$ are the only interface places in the GSPN framework where tokens are removed instead of added. Initially, place $Disabled_v$ contains as many tokens as v has parents of type SEQ. In other words,

$$\mathsf{m}(\varepsilon)(Disabled_v) = |\rho^{\mathsf{SEQ}}(v)|.$$

The template also contains auxiliary places $Current_i$ for each child. A token in

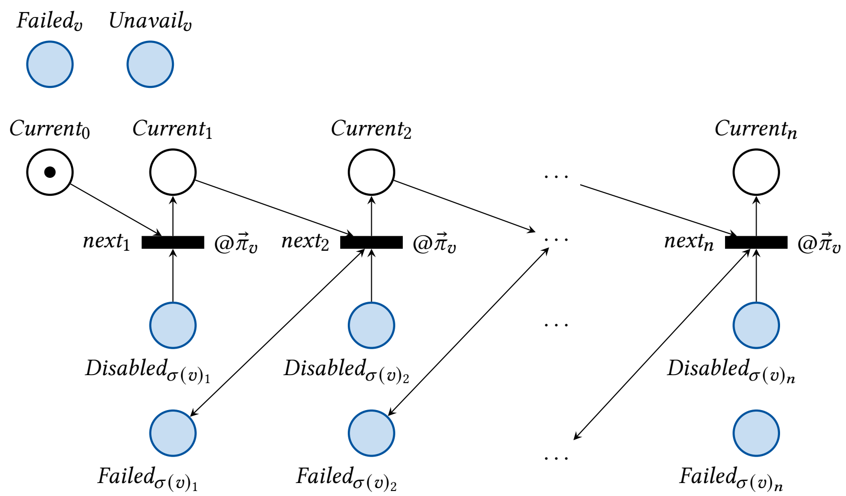


Figure 3.38: GSPN template for SEQ

$Current_i$ indicates that child $\sigma(v)_i$ can currently fail according to the order enforced by SEQ. Initially, a token is present in $Current_0$ and the immediate transition $next_1$ is enabled. Firing $next_1$ moves the token from $Current_0$ to $Current_1$ indicating that the first child can now fail. This is ensured by removing a token from $Disabled_{\sigma(v)_1}$. If the considered SEQ is the only restrictor for the first child, $Disabled_{\sigma(v)_1}$ does not contain any tokens any more. As a result, the first child can now fail. Otherwise, there are multiple restrictors governing the failure of $\sigma(v)_1$ and all of them need to allow the failure of $\sigma(v)_1$.

If a token is in $Current_i$ and the corresponding child has failed, i.e. $Failed_{\sigma(v)_i}$ contains a token, then the next child can be enabled. Transition $next_{i+1}$ fires and moves the token from $Current_i$ to $Current_{i+1}$. Additionally, a token is removed from $Disabled_{\sigma(v)_{i+1}}$. For the last child $\sigma(v)_n$ the token in $Current_n$ is consumed and not moved further.

Correctness of failure behaviour | We prove the correctness of $\mathsf{templ}_{\mathsf{SEQ}}(v)$ by showing that the interface places $Disabled_v$ capture the desired behaviour.

Theorem 3.15 ($Disabled_v$ interface place) A token is in interface place

Disabled$_v$ iff $v \in \mathsf{BE}_{\mathcal{F}}$ is *disabled*. More formally:

$$\forall v \in \mathsf{BE}_{\mathcal{F}} : \forall \tau \in \mathsf{Traces} :$$
$$v \in \mathsf{Disabled}(\tau) \iff \mathsf{m}(\tau)(\mathit{Disabled}_v) > 0.$$

Proof. We know from Lemma 3.13:

$$v \in \mathsf{Disabled}(\tau) \iff v \notin \mathsf{Failed}(\tau) \wedge \exists s \in \mathsf{SEQ}_{\mathcal{F}} : \exists 1 \leq j < i \leq n : \\ v = \sigma(s)_i \wedge \sigma(s)_j \notin \mathsf{Failed}(\tau).$$

If v is not a child of any SEQ, then $v \notin \mathsf{Disabled}(\tau)$. It also holds:

$$\mathsf{m}(\varepsilon)(\mathsf{Disabled}(v)) = |\rho^{\mathsf{SEQ}}(v)| = 0.$$

Tokens are never added to *Disabled*$_v$ by any transition. Thus, $\mathsf{m}(\tau)(\mathit{Disabled}_v) = 0$. For the other case, we assume that v is a child of a SEQ s, i.e. $v = \sigma(s)_i$. We consider both cases:

- $v \notin \mathsf{Disabled}(\tau)$: Then for all SEQs all children left of v have failed. We exemplary consider SEQ s. We know for all $1 \leq j < i$:

  $$\sigma(s)_j \in \mathsf{Failed}(\tau) \implies \mathsf{m}(\tau)(\mathit{Failed}_{\sigma(s)_j}) > 0.$$

  Thus, first transition next_1 fires and moves the token to $\mathit{Current}_1$. Afterwards, transition next_2 fires, because $\mathit{Failed}_{\sigma(s)_1}$ contains a token. The token moves to $\mathit{Current}_2$. Following this argumentation, the token eventually moves to $\mathit{Current}_{j-1}$. Firing transition next_j then removes a token from $\mathit{Disabled}_{\sigma(s)_j}$. The same reasoning holds for the other SEQs which are parents of v.

  Initially,
  $$\mathsf{m}(\varepsilon)(\mathsf{Disabled}(v)) = |\rho^{\mathsf{SEQ}}(v)| = k.$$

  All k SEQs remove one token from *Disabled*$_v$ and thus, $\mathsf{m}(\tau)(\mathsf{Disabled}(v)) = 0$.

- $v \in \mathsf{Disabled}(\tau)$: Then there exists a child left of v which is not yet failed. W.l.o.g. we assume $\sigma(s)_j$ is the leftmost child which is not failed. As all previous children have failed, we know from the previous argument that the token in the GSPN template has moved to $\mathit{Current}_j$. We know:

  $$\sigma(s)_j \notin \mathsf{Failed}(\tau) \implies \mathsf{m}(\tau)(\mathit{Failed}_{\sigma(s)_j}) = 0.$$

  Thus, transition next_{j+1} is not enabled and the token cannot move to $\mathit{Current}_{j+1}$.

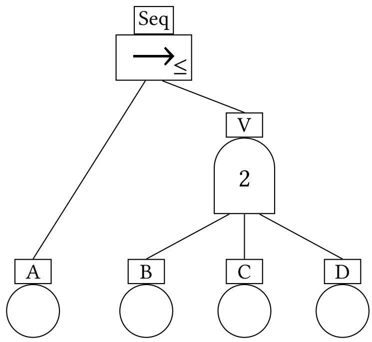


Figure 3.39: SEQ with arbitrary gates as children

As a result, transition $next_i$ was never enabled. Initially,

$$\mathsf{m}(\varepsilon)(Disabled_v) = |\rho^{\mathsf{SEQ}}(v)| \geq 1.$$

As at least the token from s was not removed from $Disabled_{\sigma(s)_i}$, it follows

$$\mathsf{m}(\tau)(Disabled_v) > 0. \qquad \blacksquare$$

3.8.1.1 Gates in SEQs

Up until now we only considered BEs as children of SEQ. That way we can perform a preprocessing to determine which BEs are disabled in the beginning. Moreover, for each disabled BE we can compute the BEs which have to fail in order to enable this BE.

If we lift DFT Constraint 13 (Inputs of SEQs are SEs) we allow arbitrary gates as children of SEQs. As a result, there might be several possibilities how a disabled BE is enabled.

Example 3.15 (SEQ with gates) We depict a $\mathsf{SEQ}_{\leq}$ with arbitrary gates as children in Figure 3.39. The $\mathsf{SEQ}_{\leq}$ has a VOT_2 gate V as second child. V is only allowed to fail if A has failed before. A valid trace in this DFT is

$$\tau = \mathsf{f}_A \oslash_{Seq} \mathsf{f}_B \oslash_{Seq} \mathsf{f}_C\ \mathsf{f}_V \oslash_{Seq}.$$

The failure of one of the children of V does not yet lead to the failure of V. Thus, one of B, C and D can fail without violating the order requirement of the SEQ.

Another valid trace therefore is

$$\tau' = \mathbf{f}_B \oslash_{Seq} \mathbf{f}_A \oslash_{Seq} \mathbf{f}_C \, \mathrm{f}_V \oslash_{Seq}.$$

An invalid trace is

$$\tau'' = \mathbf{f}_B \oslash_{Seq} \mathbf{f}_C \, \mathrm{f}_V \lightning_{Seq},$$

because here, V fails before A.

It might seem counter-intuitive at first that parts of a subtree can fail even though the failure of the complete subtree is prohibited. This behaviour however allows to model more advanced concepts, for example *mutual exclusion* (cf. Section 3.8.2).

It remains to answer the question how to handle restrictors over arbitrary gates in general. There are three possible approaches:

1. Actively prevent failures. This approach is used in the presented GSPN template. All disabled DFT elements are computed in a preprocessing step, for example by a static analysis of the DFT. In the GSPN template, place $Disabled_v$ together with inhibitor arcs is used to prevent the timed transition *fail* from firing—and the immediate transition for PDEP. An element is enabled if the set of elements preventing its failure have failed before. This set of elements can be precomputed as well. However, for arbitrary gates or when allowing dependencies to children of restrictors, computing all sequences which disable and enable an element can become quite complex and is not tractable for larger systems.

2. Passively prevent failures. This approach lets the failures happen and only afterwards checks whether the corresponding trace is valid. If the trace is invalid, we *backtrack* to the maximal prefix of the trace which was valid. This approach applies a post-processing step and is conceptually simpler to implement than the preprocessing approach. For example, our implementation for DFT analysis uses this approach (cf. Section 5.2.2.2 on page 192). However, for GSPNs, backtracking requires a roll-back to a previous marking which corresponds to a valid trace. This is quite complex to model in a Petri net.

   ·backtracking

3. Ignore failures. This approach allows disabled BEs to still fail and therefore also produces invalid traces. While performing an analysis later on, such invalid traces have to be ignored. One way for example is to use *conditional probabilities* which only consider all valid traces. This approach therefore requires analysis algorithms tailored to handle invalid traces.

   Note that in this approach, children which are dependent events of a dependency are not handled correctly. According to Definition 3.20 failures of dependent children can be 'delayed' until the child is enabled. However, if failures are just ignored, the child will always fail directly after the triggering through the PDEP and therefore never 'delay' the failure. We will investigate this issue further in Section 4.2.2 on page 150.

Due to the significantly high implementation effort, we adhere to DFT Constraint 13 (Inputs of SEQs are SEs) and DFT Constraint 14 (No dependencies to inputs of SEQ) in our GSPN framework. Most common examples from the literature—for example all DFTs from FFORT [RBNS+19]—can still be modelled with these restrictions. One exception is mutual exclusion which cannot be handled. We introduce an extra GSPN template for mutual exclusion in the following.

3

3.8.2 $\boxed{\leftrightarrow}$ Mutual exclusion (MUTEX)

Mutual exclusion [BCS10] ensures that only one of a set of DFT elements can fail. It is commonly used to model for example components which can be stuck. A valve can be stuck either open or closed, but not in both positions.

We introduce the mutual exclusion gate MUTEX. A MUTEX only allows the failure of one of its children, but not more.

Definition 3.22 (Evaluation of MUTEX) Let $\mathcal{F} = (V, \sigma, \mathit{Type}, \mathit{top}, \Theta)$ be a DFT and $\tau \in \mathsf{Traces}$. The evaluation function for $v \in \mathsf{MUTEX}_{\mathcal{F}}$ is given as follows:

$$\mathsf{Eval}(\tau, v) = \begin{cases} \lightning_v & \text{if } \lightning_v \notin \tau \wedge \mathsf{Viol}(\tau, v) \\ \oslash_v & \text{otherwise.} \end{cases}$$

where the violation condition is given by:

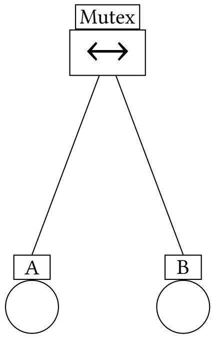
$$\mathsf{Viol}(\tau, v) \iff \exists i \neq j : \mathrm{f}_{\sigma(v)_i} \in \tau \wedge \mathrm{f}_{\sigma(v)_j} \in \tau.$$

If one child of a MUTEX has failed, all other children are disabled.

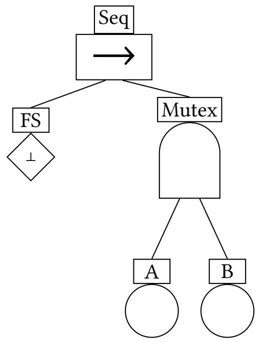
Lemma 3.16 (Disabled elements for MUTEX) Let $\mathcal{F} = (V, \sigma, \mathit{Type}, \mathit{top}, \Theta)$ be a DFT and let $\tau \in \mathsf{Traces}$.
The *disabled elements* for MUTEX are given as: ·MUTEX

$$v \in \mathsf{Disabled}(\tau) \iff v \notin \mathsf{Failed}(\tau) \wedge \exists s \in \mathsf{MUTEX}_{\mathcal{F}} : \exists v' \in \sigma(s) : v \in \sigma(s) \wedge v' \in \mathsf{Failed}(\tau).$$

The MUTEX is syntactic sugar and can be modelled with a SEQ. The construction for a binary MUTEX is given in Figure 3.40. The first child of the SEQ is a *constant fail-safe* BE. It ensures that the second child of the SEQ—an AND—is never allowed to fail. As a result, only one of the children A and B of the AND can fail. If for example A fails, B cannot fail any more, and vice versa.



(a) MUTEX



(b) SEQ modelling MUTEX

Figure 3.40: SEQ modelling a mutual exclusion MUTEX

GSPN template | We give a dedicated GSPN template for a MUTEX with n children in Figure 3.41. In the beginning, the auxiliary place *Mutex* contains a token. If one of the children $\sigma(v)_i$ fails, the corresponding immediate transition is enabled. Firing the transition consumes the token in *Mutex* and places tokens in all places $Disabled_{\sigma(v)_j}$ for the other children $j \neq i$. These children are therefore disabled. As the single token in *Mutex* is consumed, only one of the transitions can fire.

3.9 Complete translation from DFT to GSPN

We have shown the translation for each type of DFT element. It remains to show how a complete translation from a DFT to a GSPN works and to show its correctness.

3.9.1 Initialisation template

·initialisation template

One last template we need is the *initialisation template* which performs the correct initialisation of the GSPN according to the DFT structure. The initialisation template $\mathsf{templ}_{\mathsf{init}}$ is depicted in Figure 3.42. Initially, a token is present in auxiliary place *Init*. Firing transition *activate* starts the *activation*. The transition places a token in $Active_{top}$, the *Active* place of the top-level element. From there on, all elements in the *top module* are activated through the activation template (cf. Figure 3.29 on page 106). While activation tokens could already be included while building the GSPN, using the initialisation template offers a general approach independent of the DFT and it does not require any preprocessing. Moreover, failures on demand, i.e. through activation, can

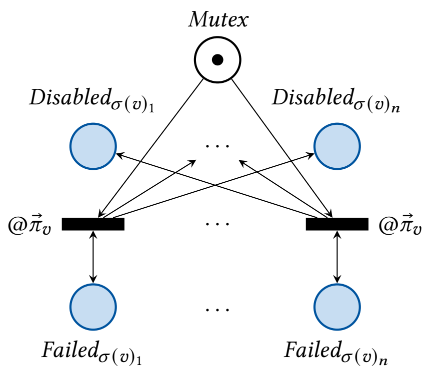


Figure 3.41: GSPN template for MUTEX

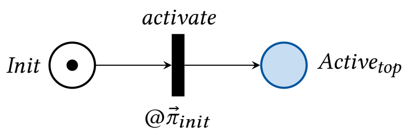


Figure 3.42: GSPN template for initialisation

be concisely modelled within this framework. The order of such failures is determined by the priority variables given to the activation transitions.

3.9.2 Complete GSPN template

We are finally able to define the complete translation of a DFT into a GSPN.

Definition 3.23 (GSPN template for DFT) Let $\mathcal{F} = (V, \sigma, Type, top, \Theta)$ be a DFT. The *GSPN template $\mathcal{G}_{\mathcal{F}}$ for DFT $\mathcal{F}$* over *interface places $\mathcal{I}_{\mathcal{F}}$* and *priority variables $\vec{\pi}$* is defined by: ·for DFT

$$\mathcal{G}_{\mathcal{F}} = \mathsf{merge}(T)$$

where the set of templates T is given by

$$T = \left\{\mathsf{templ}_{Type(v)}(v) \mid v \in \mathcal{F}\right\} \cup \left\{\mathsf{templ}_{\mathsf{act}}(v) \mid v \in \mathsf{ModuleRepr}\right\} \cup \left\{\mathsf{templ}_{\mathsf{init}}\right\}.$$

The correctness of the GSPN template follows from the results shown throughout this chapter and is based on the interface places.

Theorem 3.17 (Correctness of DFT to GSPN translation) Let $\mathcal{F}$ be a DFT and $\mathcal{G}_{\mathcal{F}}$ the corresponding GSPN template.
The *interface places* of GSPN $\mathcal{G}_{\mathcal{F}}$ capture the *state* of DFT $\mathcal{F}$. For all traces $\tau \in$ Traces it holds:

- For all $v \in \mathcal{F}$:

$$\begin{aligned} v \in \mathsf{Failed}(\tau) &\iff \mathsf{m}(\tau)(\mathit{Failed}_v) > 0 \\ v \in \mathsf{Active}(\tau) &\iff \mathsf{m}(\tau)(\mathit{Active}_v) > 0 \\ v \notin \mathsf{Available}(\tau) &\iff \mathsf{m}(\tau)(\mathit{Unavail}_v) > 0. \end{aligned}$$

- For all $v \in \mathsf{SPARE}$ and all $v' \in \sigma(v)$:

$$v' \in \mathsf{Claimed}(\tau)(v) \iff \mathsf{m}(\tau)(\mathit{Claimed}^v_{v'}) > 0.$$

- For all $\in \mathsf{BE}_{\mathcal{F}}$:

$$v \in \mathsf{Disabled}(\tau) \iff \mathsf{m}(\tau)(\mathit{Disabled}_v) > 0.$$

Proof. Correctness of the interface places follows from the corresponding theorems:

Failed: Theorem 3.1 on page 61 and the proofs for each DFT element throughout this chapter,

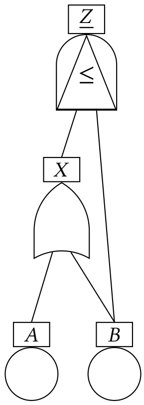
Active: Theorem 3.2 on page 62,

Unavail: Theorem 3.11 on page 96,

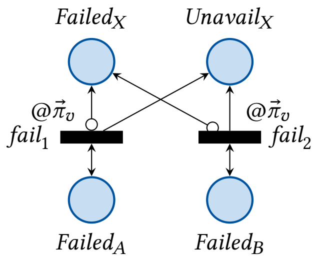
Claimed: Theorem 3.12 on page 99,

Disabled: Theorem 3.15 on page 119. ∎

Example 3.16 (DFT to GSPN translation) We present the translation of the DFT in Figure 3.43a into a GSPN template. Each of the four DFT elements A, B, X and Z is translated into their respective GSPN templates. We exemplary depict the resulting GSPN template $\mathsf{templ}_{\mathsf{OR}}(X)$ for OR X in Figure 3.43b. Combining the four GSPN templates with the activation template(s) and the initialisation template



(a) Example DFT (repeated from Figure 3.1a)



(b) GSPN template for OR X

Figure 3.43: Example translation from DFT to GSPN

yields the complete GSPN template:

$$\mathcal{G}_{\mathcal{F}} = \mathrm{merge}\Big(\mathrm{templ}_{\mathrm{PAND}_{\leq}}(Z), \mathrm{templ}_{\mathrm{OR}}(X), \mathrm{templ}_{\mathrm{BE}}(A), \mathrm{templ}_{\mathrm{BE}}(B),$$
$$\mathrm{templ}_{\mathrm{act}}(Z), \mathrm{templ}_{\mathrm{init}}\Big).$$

The complete GSPN template is depicted in Figure 3.44. Each box represents one template. To avoid overlapping, we omit the boxes of the activation templates. The resulting GSPN can be simplified by removing irrelevant places and transitions. For example, the places $Disabled_A$ and $Disabled_B$ are irrelevant as the DFT does not contain any restrictors. Similarly, all *Unavail* places can be removed because no SPARE gates are present in the DFT. Furthermore, both BEs will be activated in the beginning, because they are part of the top module. That means, we can directly place the tokens in the corresponding *Active* places instead of firing the immediate transitions in the beginning. As a result, the complete activation and initialisation templates can be omitted. In the end, we obtain the simplified GSPN template as depicted in Figure 3.45. Note that even further simplification is possible, cf. Figure 3.1c on page 57.

Before analysing a complete GSPN template $\mathcal{G}_{\mathcal{F}}$, the priority variables $\vec{\pi}$ need to be instantiated and the partitioning $\mathcal{D}$ needs to be defined. Depending on these choices, the immediate transitions in the GSPN can be fired in very different orders. These different orders represent the different evaluation orders for the DFT.

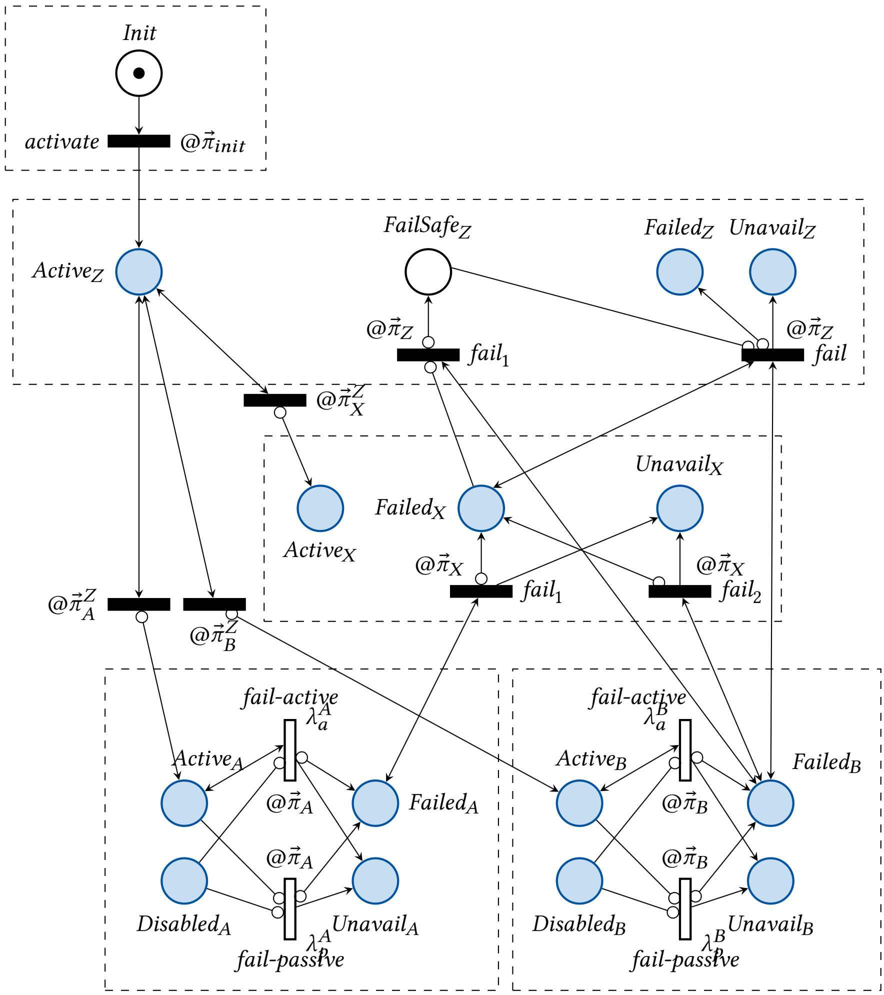


Figure 3.44: Complete GSPN template for DFT

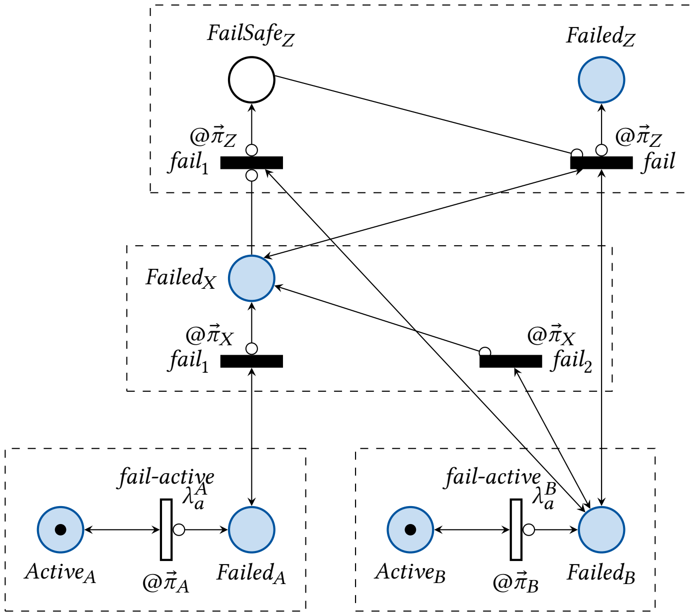


Figure 3.45: Simplified GSPN template

Tool support | The presented translation from a DFT into a GSPN is implemented in the probabilistic model checker Storm [2] and publicly available online[1]. Given a DFT, Storm performs the compositional translation and obtains the complete GSPN. During the translation, simplifications are performed by omitting unnecessary places and transitions. Examples for such simplifications were given in Example 3.16. If for example no SEQs are present, all places $Disabled_v$ can be omitted from the resulting GSPN. The obtained GSPN can be exported and used as input for dedicated GSPN analysis tools such as GreatSPN [ABBD+16]. Alternatively, the GSPN can be be translated into the JANI language [BDHH+17] which is a common input format for probabilistic model checkers.

3.10 Properties of resulting GSPNs

We conclude this chapter by giving some helpful properties of the generated GSPNs. Let $\mathcal{G}_{\mathcal{F}} = (P, T, I, O, H, m_0, W, \Pi, \mathcal{D})$ be the GSPN obtained from the DFT $\mathcal{F} = (V, \sigma, \mathit{Type}, \mathit{top}, \Theta)$.

[1] https://www.stormchecker.org/publications/gspn-semantics-for-dfts

We start by showing that the size of the resulting GSPN is linear in the size of the original DFT. We denote the maximal number of children in $\mathcal{F}$ by $\sigma_{max} = \max_{v \in \mathcal{F}} |\sigma(v)|$.

Lemma 3.18 (Size of GSPN) GSPN $\mathcal{G}_{\mathcal{F}}$ has at most $9 \cdot |V| \cdot \sigma_{max}$ places, $2 \cdot |\mathrm{BE}_{\mathcal{F}}|$ timed transitions and $6 \cdot |V| \cdot \sigma_{max}$ immediate transitions.

Proof (sketch). We give the precise number of places, transitions and arcs for each GSPN template and use this to bound the size of the GSPN. The complete proof is given in Appendix C.1.1 on page 349. ■

Next, we show that tokens in interface places are never removed—except for *Disabled*. That allows us to use the interface places to specify the current state of the DFT. Additionally, it shows that *monotonicity* of failures is preserved in the GSPN.

Lemma 3.19 (Tokens stay in interface places) In GSPN $\mathcal{G}_{\mathcal{F}}$, for all interface places $p \in \mathcal{I} \setminus \{Disabled_v \mid v \in \mathrm{BE}_{\mathcal{F}}\}$, the tokens placed in p are never removed:

$$\forall m \in M : m(p) > 0 \implies m'(p) > 0,$$

where m' is a marking reachable from m.

Proof (sketch). For each GSPN template we give a set of places P', where each transition $t \in T$ which removes a token from $p \in P'$ also puts a token back in p again:

$$I(t)(p) = 1 \implies O(t)(p) = 1.$$

We then have $P' \supseteq \mathcal{I} \setminus \{Disabled_v \mid v \in \mathrm{BE}_{\mathcal{F}}\}$. The complete proof is given in Appendix C.1.2 on page 350. ■

Each transition which puts a token in $Failed_v$ also puts a token in $Unavail_v$.

Lemma 3.20 (*Unavail*$_v$ mimics *Failed*$_v$) For all reachable marking m and all DFT element $v \in \mathcal{F}$ it holds:

$$m(Failed_v) > 0 \implies m(Unavail_v) > 0.$$

Proof. From Lemma 3.19 we know that tokens in $Failed_v$ are never removed. We therefore only consider the transitions t with $I(t)(Failed_v) = 0$, where the number of tokens can increase. By construction, we have in each template for such transitions:

$$O(t)(Failed_v) = 1 \implies O(t)(Unavail_v) = 1. \qquad ■$$

Lemma 3.21 (Transitions fire at most once) Each transition fires at most once.

Proof (sketch). For each GSPN template we define a mapping $\mathsf{M}(t) = p$ where place $p \in P$ prevents multiple firings of $t \in T$ by an inhibitor arc. Additionally, tokens in p are never removed. For some GSPN templates we have to give specific arguments like tokens moving through the system. The complete proof is given in Appendix C.1.3 on page 352. ■

Corollary 3.22 (No timeless trap) $\mathcal{G}_{\mathcal{F}}$ contains no *timeless trap*.

Proof. By Lemma 3.21 each transition in $\mathcal{G}_{\mathcal{F}}$ is fired at most once. Therefore the *reachability graph* $\mathsf{RG}(\mathcal{G}_{\mathcal{F}})$ is cycle-free and no infinite firing sequence is possible. ■

Let $Collect_{max} = \max_{v \in \mathsf{VOT}_k} |\sigma(v)|$ and let $Disabled_{max} = \max_{v \in \mathsf{BE}} |\rho^{\mathsf{SEQ}}(v)|$.

Lemma 3.23 (GSPNs are bounded) The GSPN $\mathcal{G}_{\mathcal{F}}$ is bounded by

$$\max \{2, Collect_{max}, Disabled_{max}\} .$$

All places $P \setminus (\{Unavail_v \mid v \in \mathcal{F}\} \cup \{Collect \mid v \in \mathsf{VOT}_k\} \cup \{Disabled_v \mid v \in \mathsf{BE}\})$ are 1-bounded.

Proof (sketch). We consider each GSPN template in detail and show the boundedness for each place. Most 1-bounded places have inhibitor arcs which prevent adding multiple tokens. The complete proof is given in Appendix C.1.4 on page 353. ■

In the next chapter we investigate the behaviour of complete DFTs and present several semantic intricacies observable from our GSPN framework. We also look at different semantics from the literature and compare them using the GSPN framework.

Summary

- ➜ We defined the behaviour of each DFT element as their output action for a given input trace. The order in which elements are evaluated and observed is specified by the evaluation order. Different orders can lead to significantly different behaviours of the DFT.
- ➜ We presented a compositional framework to translate a DFT into a GSPN using GSPN templates for each DFT element type.
- ➜ We proved the correspondence between traces in the DFT and firing sequences in the GSPN as well as states in the DFT and markings in the GSPN. The evaluation order in the DFT corresponds to priorities in the GSPN.
- ➜ The resulting GSPN is bounded, contains no timeless traps and each transition fires at most once.
- ➜ Semantics for priority gates PAND and POR needs to solve two questions:
  - Are simultaneous failures allowed? This is solved by inclusive and exclusive gates.
  - Does the priority order need to respect causal ordering as well? This is solved by adhering to the observed trace order induced by the evaluation order.
- ➜ SPARE-gates combine two mechanisms: claiming and activation. Claiming ensures exclusive use of a spare element. Activation changes the mode of a component from passive to active behaviour.
- ➜ Spare races are possible whenever multiple SPAREs try to claim the same shared spare child. Spare races can be prevented by syntactic restrictions on the DFT.
- ➜ Dependency PDEP forwards failure of triggering element to dependent BEs.
- ➜ Restrictor SEQ disallows certain sequences of failures. SEQ also has inclusive and exclusive variants.
- ➜ Complex structures under SEQs require involved pre-computation of disabled elements and a backtracking approach is more promising.

4 Semantic Issues in DFTs

Assumptions

The DFT is *well-formed* and satisfies the following additional constraints:

- DFT Constraint 8 (Independence of modules)
- DFT Constraint 11 (Dependent events of PDEPs are BEs)

We have defined the semantics for each DFT element in isolation in the previous Chapter 3. This specification is not enough though. To fully define and understand the behaviour of a complete DFT, the interactions between different DFT elements must also be thoroughly examined. In this chapter, we investigate a broad range of semantic intricacies and show that there are common pitfalls and misconceptions when it comes to the details of DFTs.

We start by evaluating the expressiveness of priority gates. Additionally, we show that the commonly used representations of FDEPs—as OR-gates—and SEQs—as cold SPAREs—are incorrect in general. Both FDEP and SEQ allow unique behaviours which cannot be mimicked by other gates. Combining DFT elements in a DFT can lead to interesting scenarios with complex semantic questions. We present two examples of such behaviour: nested SPARE-gates and the combination of SEQs and FDEPs.

The major contribution in this chapter is a detailed examination of existing (state-based) DFT semantics from the literature. We compare six semantics in full detail: supported elements, imposed restrictions on the modelled DFT, tool support, etc. For all semantics, we investigate the handling of three semantic issues: *failure propagation*, *failure forwarding* through dependencies and possible *non-determinism*. We show that our GSPN framework captures all other DFT semantics by instantiating the *priority variables* and setting the *partitioning* accordingly. The GSPN framework therefore provides a clear model to compare DFT semantics and pinpoint their subtle differences.

4.1 Expressiveness of gates

We first investigate the expressiveness of the DFT elements as defined in the previous Chapter 3. We already saw in Lemma 3.4 on page 69 that AND and OR can be expressed by VOT_k. The PDEP_p models an FDEP for $p = 1$, cf. Section 3.7 on page 108.

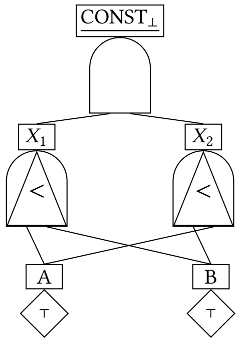


Figure 4.1: Expressing CONST$_\perp$ by PAND$_<$ [Jun15, Figure 3.6a]

As seen in Figure 3.40 on page 124, the MUTEX can be expressed using SEQ and AND. Moreover, a CONST$_\perp$ can be modelled by ensuring that a priority gate is always fail-safe. A possible construction using PAND$_<$ and CONST$_\top$ is depicted in Figure 4.1.

4.1.1 Expressiveness of Priority Gates

·expressiveness

We start by shortly considering the *expressiveness* of priority gates. The relevant insights were already developed before in [Jun15, Section 3.3.4.6] and [JGKS16] and we recapture them here for completeness. Note that the following equivalences do not hold for all possible evaluation orders, but only for bottom-up evaluation orders. A *bottom-up* order ensures that gates in the DFT are evaluated from bottom to top. That means, in each propagation step, each gate is only evaluated after all its children have been evaluated before. The restriction to bottom-up orders is not limiting, as we will argue in Section 4.4.1 that it is the 'natural' order for DFTs. We only consider binary gates here as more children can be represented by a cascade of priority gates [JGKR+17].

SAND

Simultaneous failures | The simultaneous failure of events is sometimes explicitly expressed by a dedicated gate. The *simultaneous AND gate (SAND)* fails only if all its inputs fail at the same time. The SAND is syntactic sugar and the binary SAND can be expressed by two inclusive PAND$_\leq$ as shown in Figure 4.2. PAND$_\leq$ X_1 ensures that the child A fails before or simultaneous with B, i.e. $\mathrm{f}_B \not\blacktriangleleft^{\tau}_{X_1} \mathrm{f}_A$. Similarly, PAND$_\leq$ X_2 ensures that B fails before or simultaneous with A, i.e. $\mathrm{f}_A \not\blacktriangleleft^{\tau}_{X_2} \mathrm{f}_B$. As only the simultaneous failure of A and B renders both PAND$_\leq$ failed, the AND-gate $SAND$ encodes the failure condition of an SAND. A possible trace is

$$\tau = \oslash_{SAND}\ \oslash_{X_1}\ \oslash_{X_2}\ \mathrm{f}_A\ \mathrm{f}_B\ \mathrm{f}_{X_1}\ \mathrm{f}_{X_2}\ \mathrm{f}_{SAND}.$$

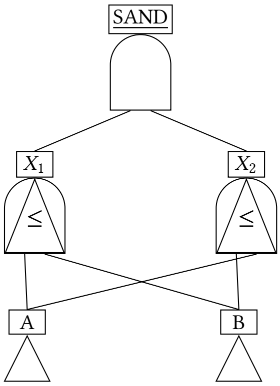


Figure 4.2: Expressing SAND by $\text{PAND}_{\leq}$ [Jun15, Figure 3.23a]

The following trace leads to wrong behaviour as it does not follow a bottom-up order:

$$\tau' = \oslash_{SAND}\ \oslash_{X_1}\ \oslash_{X_2}\ \mathrm{f}_A\ \oslash_{X_1}\ \oslash_{X_2}\ \mathrm{f}_B\ \mathrm{f}_{X_1}\ \oslash_{X_2}\ \oslash_{SAND}.$$

PAND X_2 is evaluated before its child B and thus, the PAND becomes fail-safe. As a result, the TLE $SAND$ does not fail even though the trace order observed by $SAND$ satisfies the simultaneous failures.

Expressing priority gates | Both priority gates $\text{PAND}_{\leq}$ and $\text{POR}_{\leq}$ can be expressed by the other, respectively.

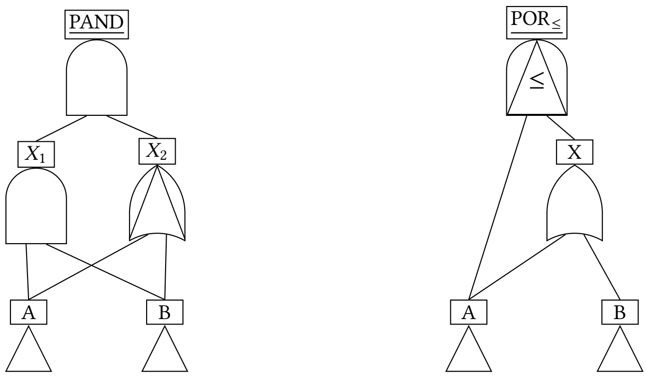
Figure 4.3a shows how to *express a PAND by a POR*. The AND X_1 ensures that both children have failed and the POR X_2 ensures that child A fails before B. In summary, AND-gate $PAND$ ensures both failure condition of a PAND. Note that if the used POR is an inclusive $\text{POR}_{\leq}$, then the encoded PAND is an inclusive $\text{PAND}_{\leq}$. Similarly, using $\text{POR}_{<}$ defines a $\text{PAND}_{<}$, because simultaneous failures do not render X_2 failed. ·express by POR

Figure 4.3b shows how to *express an inclusive $\text{POR}_{\leq}$ by an inclusive $\text{PAND}_{\leq}$*. The OR-gate X ensures that a failure of the first input A leads to a simultaneous failure of both inputs of $POR_{\leq}$. In that case, $\text{PAND}_{\leq}$-gate $POR_{\leq}$ is failed. Note that a simultaneous failure of B and A also renders $POR_{\leq}$ failed. If B fails strictly before A, i.e. $\mathrm{f}_B \blacktriangleleft^{\tau}_{X} \mathrm{f}_A$, only the second input X of $POR_{\leq}$ is failed and thus, gate $POR_{\leq}$ becomes fail-safe. Incorrect evaluation orders can render this construction invalid as well: ·express by PAND

$$\tau = \oslash_{POR_{\leq}}\ \mathrm{f}_B\ \oslash_{POR_{\leq}}\ \mathrm{f}_X\ \mathrm{f}_A\ \mathrm{f}_{POR_{\leq}}.$$

The observed trace order is $\mathrm{f}_B \blacktriangleleft^{\tau}_{POR_{\leq}} \mathrm{f}_A$ but the TLE still fails.

In contrast to the construction for a PAND it is not possible to express an exclusive



(a) Expressing PAND by POR (b) Expressing $\text{POR}_\leq$ by $\text{PAND}_\leq$

Figure 4.3: Expressing PAND and POR by each other [Jun15, Figures 3.23b and 3.23c]

$\text{POR}_<$ by an exclusive $\text{PAND}_<$ [Jun15]. It was also shown in [Jun15, Section 3.3.4.6] that the exclusive $\text{POR}_<$ can express the SAND. Thus, $\text{POR}_<$ can express all other priority gates $\text{POR}_\leq$—via $\text{POR}_<$ and SAND—and $\text{PAND}_<$ and $\text{PAND}_\leq$—both via the construction in Figure 4.3a. Thus, when using DFTs—and bottom-up evaluation orders—it suffices to support $\text{POR}_<$ as the remaining priority gates are syntactic sugar.

4.1.2 Expressiveness of FDEP

One common misconception in DFT analysis is that an FDEP is syntactic sugar and can be replaced by an OR.

> 'If there is no dependence loop in the system caused by FDEP gates, each FDEP gate is equivalent with an OR gate with two input events: one is the dependent event itself whereas the other one is the trigger event.' ([GS18])

This equivalence is assumed throughout the DFT literature, see [SVDF+02; MRL10; RS15; ZZLL11] for examples. We will show in the following though that the equivalence of FDEP and OR is only true in very restricted DFTs.

The common translation rule for FDEPs is depicted in Figure 4.4. We start with the FDEP in Figure 4.4a which triggers the failure of BE B if the subtree A has failed. The failure of B can have two reasons: either it fails on its own or it is triggered by the failure of A. We therefore can also model the two possibilities by an OR-gate as depicted in Figure 4.4b. The OR-gate B_{total} corresponds to BE B in the original DFT, i.e.

(a) FDEP (b) OR

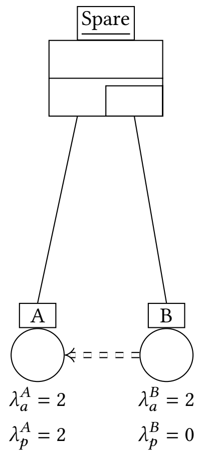
Figure 4.4: Translation from FDEP to OR

all gates which used B as input now use B_{total} instead. The OR fails if either B fails on its own or the subtree A fails. This translation rule was formalized in [JGKR+17, Rewrite Rule 24] and proven correct given some context restrictions and specific evaluation orders. In the following we show scenarios where the translation from FDEP to OR is not correct.

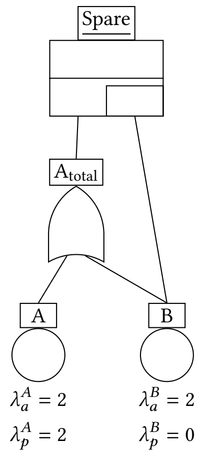
Violating DFT constraints | Replacing the FDEP by an OR can violate some of the DFT constraints given in Section 2.2.3 on page 25. For example, simply replacing *cyclic* dependencies (cf. Definition 3.19 on page 109) by OR leads to a cyclic DFT which is not well-formed any more. One workaround is to translate an SFT with cyclic dependencies into multiple SFTs without FDEPs [XDM09]. However, this approach does not work in the presence of dynamic gates. Another approach is a careful examination and manual replacement of the dependencies. However, this approach is hard to automate and might not work in all cases.

Also remember that dependencies are excluded from module paths (cf. Definition 2.16 on page 27). As a result, FDEPs can connect DFT elements which are in different modules. Replacing an FDEP by an OR connects the DFT elements by a module path, because the path can now go through the OR. As a result, elements which were previously in different modules are in the same module after the translation. Thus, DFT Constraint 8 (Independence of modules) might be violated after the translation. An example for this issue is given in Figure 4.5. In the original DFT given in Figure 4.5a, A and B belong to different modules. After the translation—given in Figure 4.5b—A and B are now part of the same module and for example claiming is not clearly defined any more.

Changed activation behaviour | The issue with merged modules due to the FDEP translation also has another effect: activation. Consider again Figure 4.5. In the original DFT, B cannot fail in the beginning, because it is not active yet. In the translated DFT



(a) Original DFT



(b) DFT after FDEP translation

Figure 4.5: Translation from FDEP to OR under a SPARE

however, B is connected to the primary child and activated in the beginning. Thus, it can already fail. The translation therefore can also change the activation behaviour.

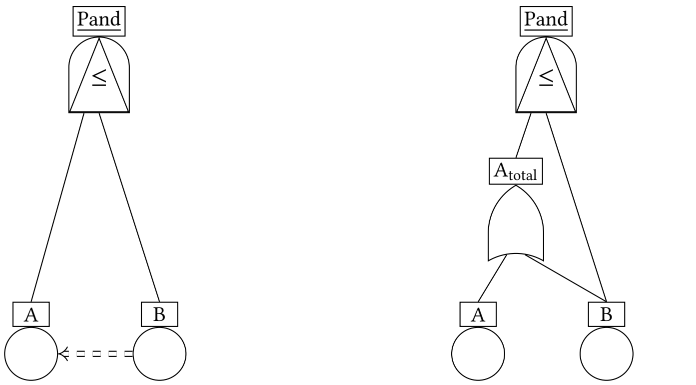
Evaluation orders | The *evaluation order* defined by the semantics can also lead to different behaviours in the translated DFT compared to the original one.

Example 4.1 (FDEP translation under PAND) We consider the DFTs with a PAND$_{\leq}$ as TLE in Figure 4.6. We use the ftrace $\boldsymbol{\tau}_\mathbf{f} = \mathbf{f}_B$ for both the original and the translated DFT. We assume a semantics where FDEPs are always evaluated after all other gates.
In the original DFT in Figure 4.6a, the trace is

$$\tau_1 = \mathbf{f}_B \oslash_{Pand} \mathrm{f}_A \oslash_{Pand}.$$

In τ_1, the PAND$_{\leq}$ is evaluated before the FDEP and therefore *Pand* becomes fail-safe due to the observed trace order $\mathrm{f}_B \blacktriangleleft^{\tau_1}_{Pand} \mathrm{f}_A$.
In the translated DFT given in Figure 4.6b, the possible traces are:

$$\tau_2 = \mathbf{f}_B\ \mathrm{f}_{A_{total}}\ \mathrm{f}_{Pand}$$
$$\tau_3 = \mathbf{f}_B \oslash_{Pand} \mathrm{f}_{A_{total}} \oslash_{Pand}$$



(a) Original DFT (b) DFT after FDEP translation

Figure 4.6: Translation from FDEP to OR under a $\text{PAND}_{\leq}$

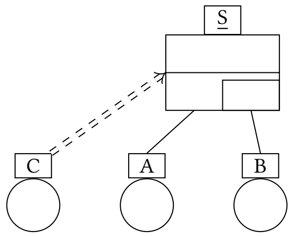
> Trace τ_3 corresponds to trace τ_1 in the original DFT. However, trace τ_2 is also valid and here the $\text{PAND}_{\leq}$ fails.

The FDEP translation is only correct if the semantics does not evaluate FDEP differently than other gates. In particular, evaluation orders which evaluate FDEPs before or after all other gates yield different results on the translated DFT compared to the original one. We rigorously examine the role of FDEP forwarding in Section 4.4.2.

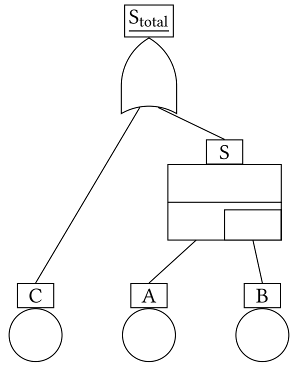
FDEPs and SEQs | If the FDEP triggers a BE under a SEQ, the failure of the BE might be ‘delayed’, because the BE is not allowed to fail yet. This behaviour is unique to FDEPs and gets lost after translation into an OR. We will investigate the role of FDEPs and SEQs in detail in Section 4.2.2.

FDEPs triggering gates | We shortly consider the behaviour where DFT Constraint 11 (Dependent events of PDEPs are BEs) is lifted, i.e. dependent elements can be gates.

> **Example 4.2 (FDEP triggering SPARE)** In the example DFT in Figure 4.7a the failure of C triggers the failure of the complete SPARE S through an FDEP. The corresponding DFT after the translation is depicted in Figure 4.7b. We consider the ftrace $\boldsymbol{\tau}_{\mathbf{f}} = \mathbf{f}_C\ \mathbf{f}_A$. A corresponding trace in the original DFT is
>
> $$\tau_1 = \upharpoonleft_S \upharpoonleft_C \mathrm{cl}^S_A \upharpoonleft_A \mathbf{f}_C\ \mathrm{f}_S\ \mathbf{f}_A\ \oslash_S.$$



(a) Original DFT



(b) DFT after FDEP translation

Figure 4.7: Translation from FDEP triggering a gate to OR

A trace in the translated DFT is

$$\tau_2 = \uparrow_{S_{total}} \uparrow_S \uparrow_C \mathrm{cl}_A^S \uparrow_A \mathbf{f}_C\, \mathrm{f}_{S_{total}}\, \mathbf{f}_A\, \mathrm{cl}_B^S \uparrow_B.$$

In the original DFT, the complete SPARE S fails and therefore cannot claim and activate children any more. However, in the translated DFT, the SPARE S itself is not failed. That means, S can still claim and activate children even though the trigger C is failed.

We assume that a gate which is triggered by an FDEP fails and therefore does not perform any actions any more. In the translated DFT however, gates are not failed but only the corresponding OR-gate which captures both failure types—internal by the gate itself and external by the FDEP. As a result, the gate still works as before. A SPARE for example can still perform claiming and activation. In case of shared spares, a failed SPARE can still claim shared children and therefore render them unavailable for other SPAREs. As a result, other SPAREs might fail earlier then in the original DFT.

DFT restrictions for correctness | In summary, the translation from FDEP to OR is correct only in a very restricted setting. We extend the restriction of [JGKR+17, Rewrite Rule 24] and provide conditions for the correctness of the translation.

Proposition 4.1 (Conditions for correctness of FDEP to OR translation)
The FDEP to OR translation in Figure 4.4 on page 137 is correct, provided the following conditions hold:

1. Both trigger and dependent event of the FDEP are part of the *top module* and not part of a *spare module*.
2. *Cyclic* dependencies are absent.
3. DFT Constraint 11 (Dependent events of PDEPs are BEs) holds.
4. DFT Constraint 14 (No dependencies to inputs of SEQ) holds.
5. The DFT semantics defines an evaluation order which does not handle FDEPs differently than other gates.

These restrictions are for example satisfied for a sub-class of DFTs called Priority Dynamic Fault Trees which only contain static gates, FDEPs and PANDs. It was shown in [MRLB10] that replacing FDEP by OR is correct in this subclass.

Ignoring the handling of FDEPs, because they are just 'syntactic sugar', is in general incorrect and leads to either wrong and inconsistent semantics or severely restricts the expressiveness of DFTs.

4.1.3 Expressiveness of SEQ

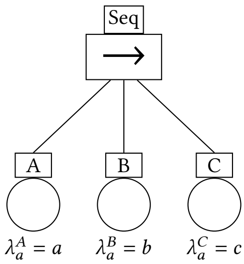
Another common misconception is that a SEQ is syntactic sugar.

> 'A fourth gate called "Sequence Enforcing" gate [...] can be emulated using a cold spare gate.' ([BCS10])

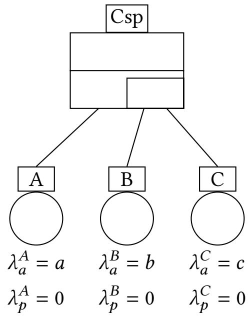
This misconception is again stated throughout the DFT literature, see [MPBV+06; RS15; GLYZ+15a; Kab17] for examples. Similar to the FDEP, we will again show that the translation from SEQ to cold SPARE is only true in a limited amount of scenarios. All issues—apart from the last one (triggering in SEQ)—were already identified in [Jun15, Section 3.3.4.8] and [JGKS16]. Some of the issues were also stated in [GS18].

The common translation rule for SEQs is depicted in Figure 4.8. We start with a SEQ with multiple SEs as given in Figure 4.8a. The translation replaces the SEQ with a *cold SPARE* as depicted in Figure 4.8b. The cold SPARE sets the passive failure rates of all children to zero. That means they can only fail if they are activated. In the beginning, BE A will be active and therefore can fail. The other children B and C are passive and therefore cannot fail. After the failure of A, BE B becomes active and can fail from now on. The failure restriction of the SEQ is imitated by the cold SPARE.

Example 4.3 (SEQ translation) We consider possible traces for both DFTs in Figure 4.8 and start with the ftrace $\tau_f = \mathsf{f}_A\ \mathsf{f}_B\ \mathsf{f}_C$. A corresponding valid



(a) SEQ



(b) Cold SPARE

Figure 4.8: Translation from SEQ to cold SPARE

trace in the original DFT is

$$\tau_{SEQ} = \upharpoonleft_A \upharpoonleft_B \upharpoonleft_C \mathbf{f}_A \oslash_{Seq} \mathbf{f}_B \oslash_{Seq} \mathbf{f}_C \oslash_{Seq}.$$

A similar valid trace in the translated DFT is

$$\tau_{CSP} = \upharpoonleft_{Csp} \mathrm{cl}_A^{Csp} \upharpoonleft_A \mathbf{f}_A \mathrm{cl}_B^{Csp} \upharpoonleft_B \mathbf{f}_B \mathrm{cl}_C^{Csp} \upharpoonleft_C \mathbf{f}_C \mathrm{f}_{Csp}.$$

A difference between both DFTs can be seen when looking at an invalid trace. We consider the ftrace $\boldsymbol{\tau}'_{\mathbf{f}} = \mathbf{f}_B \mathbf{f}_A$. The corresponding trace in the original DFT is

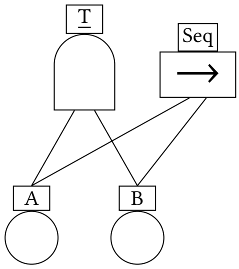
$$\tau'_{SEQ} = \upharpoonleft_A \upharpoonleft_B \upharpoonleft_C \mathbf{f}_B \lightning_{Seq} \mathbf{f}_A,$$

which is an invalid trace. A similar trace in the translated DFT would be

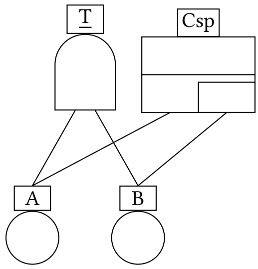
$$\tau'_{CSP} = \upharpoonleft_{Csp} \mathrm{cl}_A^{Csp} \upharpoonleft_A \mathbf{f}_B.$$

However, the BE B cannot fail in τ'_{CSP}, because B is not active and has a passive failure rate of $\lambda_p^B = 0$. Therefore, the ftrace $\boldsymbol{\tau}'_{\mathbf{f}}$ could never happen in the translated DFT as it has a zero probability of occurring.

Overlapping modules | We show the first issue by example of the simple DFT in Figure 4.9. After the translation, the SEQ is replaced by a SPARE, cf. Figure 4.9b. Due to the AND-gate, both BEs are connected via a module path (cf. Definition 2.16 on page 27). Both BEs therefore belong to the same *module* even though they are both spare representatives. Thus, DFT Constraint 8 (Independence of modules) is violated.



(a) Original DFT



(b) DFT after SEQ translation

Figure 4.9: Module issues for translation from SEQ to cold SPARE

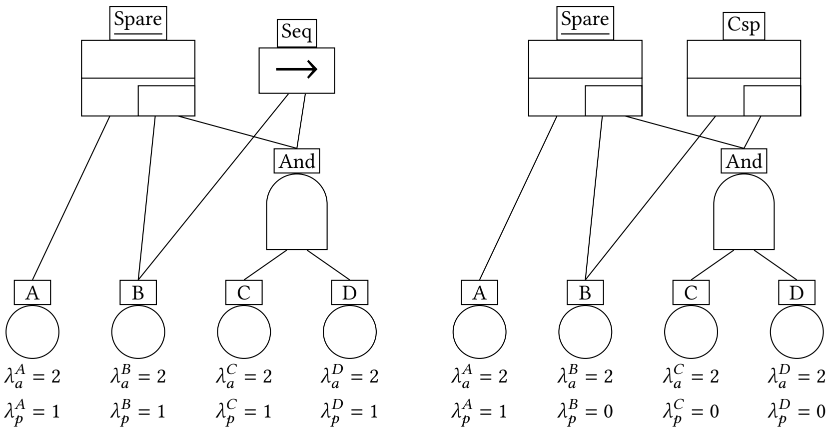
That means replacing SEQs by cold SPAREs will in general yield DFTs which are not *conventional*.

We could stop here already and state that the SEQ translation is incorrect, because it yields unsupported DFTs. However, there are approaches to circumvent this problem of overlapping modules. In the following, we show that even when ignoring DFT Constraint 8 (Independence of modules), multiple additional issues are present which render the translation incorrect in most cases.

Example 4.4 (SEQ translation) We use the example SEQ translation in Figure 4.10 as a running example. The original DFT is depicted in Figure 4.10a and consists of a SPARE and a SEQ. One of the children is an AND-gate. All four BEs have a dormancy factor of 0.5. The DFT after the translation is given in Figure 4.10b and consists of two SPARE gates.

SEQ and SPARE | The first issue occurs, when children of the SEQ are also part of a spare module. In our example, this applies to BE B for example. In the original DFT, B has a passive failure rate of 1 which is smaller than the active failure rate of 2. In particular, it means that B can also fail in passive mode. However, the translation of the SEQ into a cold SPARE requires all children to be cold BEs. Thus, B cannot fail in passive mode any more. The ftrace $\tau_\mathrm{f} = \mathbf{f}_B$ is valid in the original DFT but not in the translated DFT and therefore the translation is not correct in this instance.

SEQ over gates | In the example, we allowed gates as children of SEQs, i.e. DFT Constraint 13 (Inputs of SEQs are SEs) is lifted. The construction in Example 4.4 is similar to the construction for a MUTEX (cf. Figure 3.40 on page 124). The ftrace $\tau'_\mathrm{f} = \mathbf{f}_C$ is valid in the original DFT, because the AND-gate still stays operational. However, τ'_f is not possible in the translated DFT, because BE C cannot fail in passive mode. In



(a) Original DFT (b) DFT after SEQ translation

Figure 4.10: Issues for translation from SEQ to cold SPARE

general, gates under SEQs cannot be modelled with cold SPAREs and in particular the MUTEX-gate cannot be expressed.

Multiple SEQs | Another problem with replacing SEQs by SPAREs is that claiming is present in SPAREs. In the translated DFT of Figure 4.10b, BE B is the primary module of SPARE Csp and therefore is claimed in the beginning:

$$\tau = \upharpoonleft_{Spare} \upharpoonleft_{Csp} \mathrm{cl}_A^{Spare} \upharpoonleft_A \mathrm{cl}_B^{Csp} \upharpoonleft_B .$$

After the failure of A, $Spare$ cannot claim B, because it is already claimed by Csp. For ftrace $\tau''_{\mathrm{f}} = \mathbf{f}_A$, the corresponding trace in the translated DFT is

$$\tau' = \upharpoonleft_{Spare} \upharpoonleft_{Csp} \mathrm{cl}_A^{Spare} \upharpoonleft_A \mathrm{cl}_B^{Csp} \upharpoonleft_B \mathbf{f}_A \mathrm{cl}_{And}^{Spare} \upharpoonleft_{And} \upharpoonleft_C \upharpoonleft_D .$$

However, in the original DFT, BE B can be claimed:

$$\tau' = \upharpoonleft_{Spare} \mathrm{cl}_A^{Spare} \upharpoonleft_A \mathbf{f}_A \mathrm{cl}_B^{Spare} \upharpoonleft_B .$$

The same effect can also be observed for multiple SEQs which have the same child. After the translation, the cold SPAREs share this child, but only one of them can claim it. Again, this leads to incorrect behaviour.

Triggering in SEQ | Lastly, we consider the case where an FDEP triggers a BE under a SEQ, i.e. DFT Constraint 14 (No dependencies to inputs of SEQ) is lifted. Here, the behaviour between the original DFT and the translated one differs again. BEs which are disabled due to a SEQ cannot fail and therefore can also not be triggered by an FDEP. In contrast, even cold BEs—which cannot fail on their own—can still be triggered by an FDEP and therefore fail. Additionally, SEQs can also 'delay' the failure due to an FDEP as will be later seen in Section 4.2.2. This behaviour can not be mimicked with a cold SPARE as well and was already an issue for the FDEP translation.

DFT restrictions for correctness | In total, the translation from a SEQ into a cold SPARE is only correct in a limited amount of scenarios.

Proposition 4.2 (Conditions for correctness of SEQ translation) The SEQ to cold SPARE translation in Figure 4.8 on page 142 is correct, provided the following conditions hold:

1. DFT Constraint 8 (Independence of modules) must be ignored. A semantically sound workaround must be used to allow overlapping *spare modules*.
2. All children of the SEQ cannot be part of any *spare module*.
3. DFT Constraint 13 (Inputs of SEQs are SEs) must hold.
4. DFT Constraint 14 (No dependencies to inputs of SEQ) must hold.
5. Each BE can only be child of at most one SEQ.

Once again, SEQs need to be considered and supported, because in most cases they cannot be expressed by other gates.

4.2 Advanced behaviour in DFTs

4.2.1 Nested SPAREs

Originally, SPAREs could only have BEs as children [DBB92]. Lifting this restriction results in having arbitrary subtrees as children [BCS10] which allows for more modelling flexibility, but also introduces new semantics questions. In particular, the case where a SPARE-gate is a child of another SPARE is important to consider. We call these cases *nested SPAREs*. We recapture semantic questions here which were first considered in [Jun15] and [JGKS16]. ·nested SPAREs

4

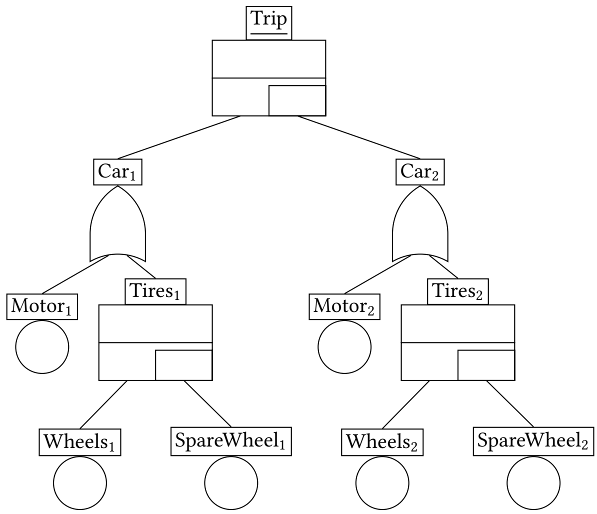


Figure 4.11: Example of nested SPAREs

Example 4.5 (Nested SPAREs) Consider the DFT in Figure 4.11 which models a car trip. The trip starts in the first Car_1. If one of the wheels of the first car fails, the car can still be used by using $SpareWheel_1$. If for example the motor fails in the first car, we have to abandon the car and switch to the second spare Car_2. The second Car_2 has the same structure as the first car and for example also comes with a spare wheel.

It is in particular interesting to look at the behaviour of the nested SPARE $Tires_2$. The question is how claiming, activation and failure work for the nested SPARE. We start with the trace prefix

$$\tau = \uparrow_{Trip} \mathrm{cl}^{Trip}_{Car_1} \uparrow_{Car_1} \uparrow_{Motor_1} \uparrow_{Tires_1} \mathrm{cl}^{Tires_1}_{Wheels_1} \uparrow_{Wheels_1}.$$

In τ, first $Trip$ is activated and claims its first child Car_1. The claiming activates the module corresponding to Car_1. In particular, $Tires_1$ is activated, and claims and activates $Wheels_1$. The spare child Car_2 is not yet claimed and therefore passive. In particular, the SPARE $Tires_2$ is also passive.

The first question is whether passive SPAREs can claim? Following [Jun15], we distinguish two possibilities. If a SPARE can claim children independent of its activation status, we call this *early claiming*. In contrast, if the claiming takes only place if the SPARE is active, we call this *late claiming*.

We define both claiming behaviours based on the condition $\mathsf{Claim}(\tau, v, \sigma(v)_i)$ given in Definition 3.15 on page 92.

Definition 4.1 (Early/Late claiming of SPARE) Let $v \in \mathsf{SPARE}_{\mathcal{F}}$, $\tau \in$ Traces, $1 \leq i \leq |v|$.
The *early claiming* of a SPARE is given by: ·early claiming

$$\mathsf{EarlyClaim}(\tau, v, \sigma(v)_i) \iff \mathsf{Claim}(\tau, v, \sigma(v)_i).$$

The *late claiming* of a SPARE is given by: ·late claiming

$$\mathsf{LateClaim}(\tau, v, \sigma(v)_i) \iff \mathsf{Claim}(\tau, v, \sigma(v)_i) \land v \in \mathsf{Active}(\tau).$$

Late claiming additionally requires that the SPARE is active. Both claiming behaviours lead to different DFT traces.

Example 4.6 (Early and late claiming) We continue with Example 4.5 and consider the ftrace $\boldsymbol{\tau_{\mathrm{f}}} = \mathrm{f}_{Wheels_2}$.
We start with early claiming. According to this definition, the passive SPARE $Tires_2$ already claims it primary module in the beginning. The corresponding trace is:

$$\tau_{ec} = \uparrow_{Trip} \mathrm{cl}^{Trip}_{Car_1} \uparrow_{Car_1} \uparrow_{Motor_1} \uparrow_{Tires_1} \mathrm{cl}^{Tires_1}_{Wheels_1} \uparrow_{Wheels_1} \mathrm{cl}^{Tires_2}_{Wheels_2} \mathrm{f}_{Wheels_2} \mathrm{cl}^{Tires_2}_{SpareWheel_2}.$$

In this trace, the unused $Tires_2$ already claims $Wheels_2$. However, $Wheels_2$ is not activated as $Tires_2$ is not active yet, cf. Definition 3.18 on page 105. After the failure of $Wheels_2$, the SPARE claims the next child $SpareWheel_2$ even though $Tires_2$ is not active.
For late claiming, passive components do not claim their components yet. The corresponding trace is:

$$\tau_{lc} = \uparrow_{Trip} \mathrm{cl}^{Trip}_{Car_1} \uparrow_{Car_1} \uparrow_{Motor_1} \uparrow_{Tires_1} \mathrm{cl}^{Tires_1}_{Wheels_1} \uparrow_{Wheels_1} \oslash_{Tires_2} \mathrm{f}_{Wheels_2} \oslash_{Tires_2}.$$

In trace τ_{lc}, $Tires_2$ has not claimed any child in the beginning. Even when its first child $Wheel_2$ fails, SPARE $Tires_2$ does not perform any claiming, because it is still not activated.

Similar to the claiming behaviour, the failure behaviour can also depend on the activation status of the SPARE. We again distinguish two behaviours for the question whether passive SPAREs can fail? If a SPARE fails regardless of its activation status, we call this *early failing*. If the failure only occurs for active SPAREs, we call it *late failing*.

We define both failure behaviours based on the condition $\mathsf{Fail}(\tau, v)$ of Definition 3.16 on page 93.

4

·early failing

·late failing

Definition 4.2 (Early/Late failing of SPARE) Let $v \in \mathsf{SPARE}$ and $\tau \in \mathsf{Traces}$. The *early failing* of a SPARE is given by:

$$\mathsf{EarlyFail}(\tau, v) \iff \mathsf{Fail}(\tau, v).$$

The *late failing* of a SPARE is given by:

$$\mathsf{LateFail}(\tau, v) \iff \mathsf{Fail}(\tau, v) \wedge v \in \mathsf{Active}(\tau).$$

Late failing again requires that the SPARE is active.

Example 4.7 (Early and late failing) We continue with Example 4.6 and consider the ftrace $\tau'_{\mathrm{f}} = \mathrm{f}_{Wheels_2}\, \mathrm{f}_{SpareWheel_2}$.
We start with early failing and early claiming. The corresponding trace is:

$$\tau_{ec/ef} = \tau_{ec}\, \mathrm{f}_{SpareWheel_2}\, \mathrm{f}_{Tires_2}\, \mathrm{f}_{Car_2}.$$

Here, the failure of $SpareWheel_2$ directly leads to the failures of $Tires_2$ and Car_2. Similarly, the trace for early failing and late claiming is given as follows:

$$\tau_{lc/ef} = \tau_{lc}\, \mathrm{f}_{SpareWheel_2}\, \mathrm{f}_{Tires_2}\, \mathrm{f}_{Car_2}.$$

Here, the failures of $Tires_2$ and $SpareWheel_2$ directly lead to the failure of the SPARE $Tires_2$. Note that the SPARE fails even though it never claimed any child. In contrast, the trace for late failing and late claiming looks as follows:

$$\tau_{lc/lf} = \tau_{lc}\, \mathrm{f}_{SpareWheel_2}\, \oslash_{Tires_2}.$$

In $\tau_{lc/lf}$, the SPARE $Tires_2$ does not fail yet, because it is passive. When it becomes active, it immediately fails, because both children failed before.
We also give the trace for late failing and early claiming:

$$\tau_{ec/lf} = \tau_{ec}\, \mathrm{f}_{SpareWheel_2}\, \oslash_{Tires_2}.$$

In this case, the SPARE does not fail even though claiming was not successful.

Note that our definition of early/late failing is conceptually slightly different compared to the original definition [Jun15]. Originally, early/late failing answers the question, when an unsuccessful claiming—and therefore the failure of the SPARE—is detected. For late claiming, it is detected that the claiming will be unsuccessful either when performing the actual claiming—corresponding to late failing—or already before claiming when the child becomes unavailable—corresponding to early failing.

In the original definition, early claiming with late failing was not possible, because

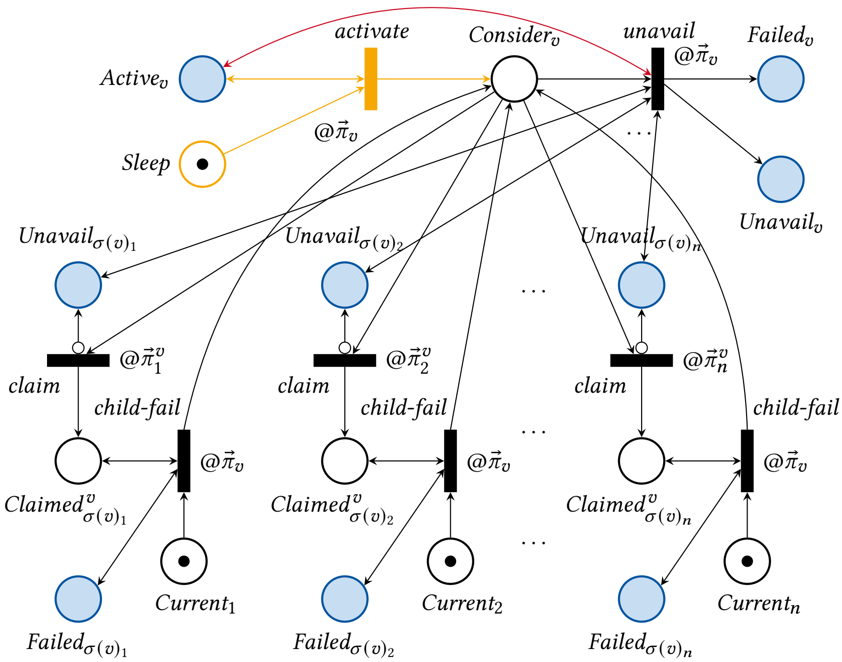


Figure 4.12: GSPN template for SPARE with late claiming (orange) and late failing (red)

'under early claiming it is clear when a SPARE fails, as it can claim whenever it wants' [JGKS16].

Given our definitions for $\mathsf{Claim}(\tau, v, \sigma(v)_i)$ and $\mathsf{Fail}(\tau, v)$, it is still possible that a SPARE stays operational as long it is not activated—even though the claiming was already unsuccessful.

The four different variants introduce specific behaviours with respect to causality. In early claiming with early failing, the SPARE only fails if one of its children also immediately failed before. In both variants of late failing, a SPARE can fail due to activation. In late claiming with early failing, a SPARE can fail because other SPAREs claimed shared spares. All three latter variants can also introduce spare races—even for independent sub-modules. For the complete details on nested SPAREs, we refer to [Jun15].

GSPN template | The four SPARE variants can all be represented by a GSPN template. The template for a SPARE with arbitrary claiming order (cf. Figure 3.26 on page 98) corresponds to early claiming and early failing. We extend this template to ensure late claiming and late failing. The resulting GSPN template is depicted in Figure 4.12.

The extension for late claiming is highlighted in orange colour. We introduce a transition *activate* which ensures that a token is only placed in $Consider_v$ if the SPARE is active, i.e. $Active_v$ contains a token.

The extension for late failing is highlighted in red colour and is just one additional arc from $Active_v$ to *unavail*. The additional arc ensures that the SPARE can only fail if it is active. By using—or not using—these extensions we can represent all four SPARE variants in our GSPN framework.

Discussion | In our opinion, two of the four nested SPARE variants are the most realistic: early claiming with early failing and late claiming with late failing. The first variant always directly claims and fails regardless of its activation status. The second variant only performs an action—claiming or failing—if it is active. In the variant early claiming with late failing, claiming can be unsuccessful, but the SPARE still stays operational. In the variant late claiming with early failing, no claiming takes places, but still the failure of the SPARE can be detected without even trying to claim. This was also called a 'kind of look ahead feature' [JGKS16]. We think that the two latter variants are only interesting in very specific scenarios. In general, a consistent behaviour of claiming and failure with respect to activation is preferable.

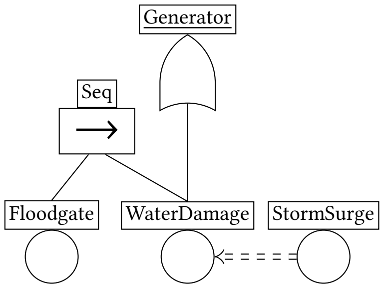
4.2.2 Restrictors and Dependencies

We shortly investigate some intricacies arising from combining restrictors with dependencies. Namely, we look at FDEPs which trigger children of SEQs.

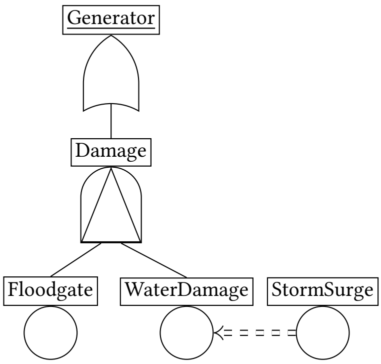
We start by giving a general motivation why FDEPs and SEQs are necessary in some modelling scenarios.

Example 4.8 (Floodgate and water damage) Consider the DFT in Figure 4.13a which models possible water damage. The TLE is a generator which fails if for example water damage occurs. Water damage can only occur if the floodgate—which prevents the water from reaching the generator—has failed before. We use a SEQ to ensure that water damage can only occur after the floodgate has failed. Water damage can occur on its own—through heavy rain for example. Another possibility is if a storm surge happens, which also leads to water damage. We model this with an FDEP, because a storm surge also triggers the failure of other parts of the system.
We consider the trace

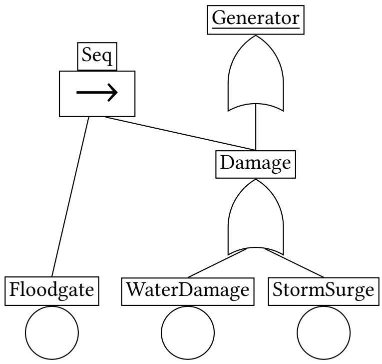
$$\tau_1 = \mathrm{f}_{StormSurge}\ \mathrm{f}_{WaterDamage}\ \lightning_{Seq}.$$



(a) SEQ and FDEP



(b) PAND and FDEP



(c) SEQ and OR

Figure 4.13: DFTs modelling water damage

This trace is invalid, because the floodgate is still operational in τ_1 and prevents the water damage through the storm surge. We explicitly prevented such failures in Definition 3.20 on page 111 by requiring that disabled elements cannot be triggered.
In the following, we consider the ftrace

$$\tau_{\mathrm{f}} = \mathrm{f}_{StormSurge}\ \mathrm{f}_{Floodgate}.$$

The following corresponding trace is valid:

$$\tau_2 = \mathrm{f}_{StormSurge}\ \oslash_{WaterDamage}\ \mathrm{f}_{Floodgate}\ \oslash_{Seq}\ \mathrm{f}_{WaterDamage}\ \oslash_{Seq}\ \mathrm{f}_{Generator}.$$

In τ_2, again a storm surge happens, but the water damage does not occur yet due to the still functioning floodgate. When the floodgate fails, the storm surge

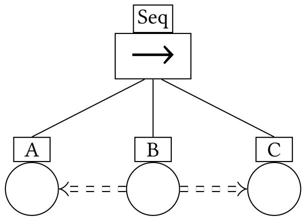


Figure 4.14: Handling of simultaneous failures in SEQ

immediately leads to the water damage and therefore the failure of the generator. In the example, the triggering of water damage through the FDEP is 'delayed' through the SEQ. This behaviour cannot be modelled by using a PAND for example. We show this in the DFT of Figure 4.13b which uses a PAND. For the same ftrace τ_f as before we get the following trace

$$\tau_3 = \mathsf{f}_{StormSurge}\ \mathsf{f}_{WaterDamage}\ \oslash_{Damage}\ \mathsf{f}_{Floodgate}\ \oslash_{Damage}.$$

An occurrence of the storm surge while the flood gate is still operational leads to a fail-safe state of the PAND. If the floodgate fails afterwards, the PAND stays fail-safe and damage still does not occur meaning the generator will not fail. In contrast, in the SEQ, the failure of the floodgates allows the occurrence of water damage and thus, the generator fails.
If delayed failure propagation should not occur, we can replace the FDEP with an OR-gate as depicted in Figure 4.13c. Here, the ftrace τ_f leads to the trace

$$\tau_4 = \mathsf{f}_{StormSurge}\ \mathsf{f}_{Damage}\ \lightning_{Seq}\ \mathsf{f}_{Floodgate},$$

which is invalid. That means, for this DFT, a storm surge cannot occur—or is at least not considered—as long as the floodgate is operational.

In summary, using FDEPs and SEQs together allows to model scenarios which cannot be modelled by other DFT elements.

Allowing FDEPs to trigger children of SEQs also requires the distinction between *inclusive* and *exclusive* SEQ.

Example 4.9 (Inclusive and exclusive SEQ) We show the difference between inclusive and exclusive SEQ for the example DFT in Figure 4.14.

We consider the ftrace $\boldsymbol{\tau_{\mathrm{f}}} = \mathbf{f}_B$. A possible trace for $\mathrm{SEQ}_{\leq}$ is:

$$\tau_1 = \mathbf{f}_B\, \mathrm{f}_A\, \oslash_{Seq}\, \mathrm{f}_C\, \oslash_{Seq}.$$

The failures of A and B are observed to be simultaneous and therefore the trace is valid. For $\mathrm{SEQ}_{<}$, a similar trace would be invalid:

$$\tau_2 = \mathbf{f}_B\, \mathrm{f}_A\, \lightning_{Seq}.$$

We also consider the ftrace $\boldsymbol{\tau'_{\mathrm{f}}} = \mathbf{f}_A\, \mathbf{f}_B$. Possible traces for $\mathrm{SEQ}_{<}$ are:

$$\tau_3 = \mathbf{f}_A\, \oslash_{Seq}\, \mathbf{f}_B\, \mathrm{f}_C\, \lightning_{Seq}$$
$$\tau_4 = \mathbf{f}_A\, \oslash_{Seq}\, \mathbf{f}_B\, \oslash_{Seq}\, \mathrm{f}_C\, \oslash_{Seq}.$$

While the same traces would both be valid for $\mathrm{SEQ}_{\leq}$, only τ_4 is valid for $\mathrm{SEQ}_{<}$. We can see that the definition of disabled elements in Lemma 3.13 on page 117 is not correct for these examples. BE B would be disabled in the beginning, even though its failure leads to the valid trace τ_1 for $\mathrm{SEQ}_{\leq}$. In τ_3 and τ_4, C is not disabled any more after the failure of B. However, for a semantics which always evaluates FDEPs before SEQs, only τ_3 would be considered. In this case, all following traces are invalid and therefore C should be disabled according to Definition 2.34 on page 41. In summary, when allowing dependencies to inputs of SEQs, the simple pre-computation of disabled elements as done in Lemma 3.13 on page 117 is not correct and more involved computations are necessary.

The problems in the previous example stem from simultaneous failures. Simultaneous failures cannot only come from FDEPs, but can also be achieved for example by common cause failures of OR-gates. Thus, both constraints for Lemma 3.13 are necessary to ensure correctness.

In summary, children of SEQs which are gates or can be triggered by FDEPs require a more involved computation of disabled elements. This computation might not be feasible as a pre-computation step, because the disabled elements for a large number of different traces need to be considered. In particular, for each possible failure, all trace suffixes need to be considered to determine whether valid traces are possible. As this approach does not scale well, other approaches such as *backtracking* are more promising and should be used for handling restrictors.

4.3 DFT Semantics from the literature

In the rest of this chapter, we investigate several DFT semantics from the literature. We consider the most prominent and commonly used state-based DFT semantics.

Table 4.1: Overview of considered DFT semantics

Semantics	Formalisation	Tool support	Underlying model
Galileo	[CSD00]	Galileo	CTMC
CR-GSPN	[Rai05]	DFT2GSPN	GSPN/CTMC
DFTCalc	[BCS10]	DFTCalc	I/O-IMC
Storm	[6]	Storm	MA
IOSA	[MBD20]	FIG	IOSA
GSPN framework	[4]	Storm	GSPN/MA

We compare them on a multitude of aspects: supported elements, constraints on the modelled DFTs, tool support, etc. We also compare the precise semantics in detail, in particular their behaviour on three semantic intricacies presented in Section 4.4. Five of the semantics were already compared in [4], semantics *IOSA* [MBD20] is added here.

4.3.1 Considered DFT semantics

We introduce the DFT semantics which we consider in the following. Note that we only consider DFT semantics here which are state-based, i.e. they use some form of automaton as underlying model. An overview of the considered semantics is given in Table 4.1. Column 'Formalisation' indicates the main paper which defines the corresponding DFT semantics. If not otherwise specified, all details of the DFT semantics are extracted from these papers.

Galileo | DFTs were first introduced by Dugan, Bavuso and Boyd [DBB90; DBB92] and the behaviour of each DFT element was informally described. As also noted by the original authors [CSD00], this specification was not precise enough in several regards and raised semantic questions. They therefore presented a precise formalisation of DFTs in [CSD00]. Additional specification details were also given in [CS98] and [MCSD99]. The specification is provided in the formal specification notation Z [Spi89] and based on so-called failure automata. The underlying model for the DFT is a *continuous-time Markov chain* (CTMC).

Tool support for DFT analysis was initially added to the HARP tool [DTSG86]. The DIFtree software package [DVG97] provided dedicated analysis methods for static and dynamic subtrees and also offered a graphical user interface. The Galileo tool [SDC99] was the successor of DIFtree and is probably the reference implementation for DFT analysis. However, it is discontinued by now and not available any more.

We identify the original DFT semantics by the name of its tool *Galileo*.

CR-GSPN | Chronologically, the next DFT semantics was presented by Raiteri [Rai05]. The semantics is given as a model-to-model transformation from DFTs to *GSPNs* using graph transformation rules. The approach is very similar to our GSPN framework presented in Chapter 3. The idea is the same: each DFT element is successively translated into a GSPN. The underlying model is thus a GSPN which itself is again translated into a CTMC.

The translation from DFT to GSPN is implemented in the converter DFT2GSPN. The DFT is drawn in the DrawNet tool [FGIM+02] and converted to a GSPN which is supplied to the GreatSPN tool[1] [ABBD+16]. GreatSPN then performs the GSPN analysis. It is not clear whether DFT2GSPN was publicly available at some point, but it is not available any more.

We identify this semantics by *CR-GSPN* (Codetta-Raiteri-GSPN) to avoid confusion with our *GSPN framework*.

IOIMC | The compositional DFT semantics by Boudali, Crouzen and Stoelinga [BCS10] was developed to lift some of the restrictions of the original *Galileo* semantics. The new semantics is based on input/output interactive Markov chains (I/O-IMC) [BCS10] which are a combination of interactive Markov chains (IMCs) [Her02] and input/output automata (I/O-automata) [LT87]. I/O-IMCs allow for parallel composition via synchronization on input and output actions and support non-determinism. As a result, the resulting model can be a continuous-time Markov decision process (CTMDP) [Ber05]. In practice however, the resulting model most often is a *CTMC*, because the non-determinism is *spurious*.

Tool support was first given in the tool CORAL [BCS07] and later implemented in the dedicated DFT analysis tool DFTCalc[2] [ABBG+13].

We identify this semantics by its tool *DFTCalc*.

Storm | We provide our DFT semantics in [6] which is based on the *Galileo* semantics but with less restrictions. One difference is also that the semantics is based on *Markov automata (MA)* and therefore allows non-determinism.

The DFT analysis is implemented in the probabilistic model checker Storm[3] [2]. Details on the implementation will be provided later in Chapter 7.

We denote this semantics by its tool *Storm*.

IOSA | Another compositional semantics was recently presented by Monti, Budde and D'Argenio [MBD20]. It is based on input/output stochastic automata (IOSA) [DM18] which combine continuous probability jumps present in stochastic automata [DK05]

[1] https://github.com/greatspn/SOURCES
[2] https://fmt.ewi.utwente.nl/tools/dftcalc/
[3] https://www.stormchecker.org/

with discrete event synchronization. IOSA allows for compositional modelling and supports arbitrary continuous probability distributions. One main property of the DFT semantics is that it leads to weakly deterministic models [MBD20] and therefore only *spurious* non-determinism is present. The focus of the semantics presented in [MBD20] is on *repairable DFTs*, but we only consider the non-repairable fragment in the following.

Tool support is provided by a toolchain [BBMD+20] using the rare event simulator FIG [Bud20]. Note that discrete event simulation as implemented in FIG can be applied, because the resulting models are weakly deterministic.

We identify this semantics by its underlying model *IOSA*.

GSPN framework | The last semantics is the *GSPN framework* introduced in [4] and presented in Chapter 3. Strictly speaking, it is not a unique semantics, because it tries to subsume the existing semantics. Similar to *CR-GSPN*, the *GSPN framework* is based on *GSPNs* but the underlying model is a *Markov automaton* instead of a CTMC following the approach in [EHKZ13]. Using Markov automata allows to incorporate non-determinism and therefore offers greater modelling flexibility.

Tool support is provided in Storm[4] [2] which translates a DFT into a GSPN. The resulting GSPN can either be analysed by dedicated analysis tools such as GreatSPN [ABBD+16] or can be translated into the JANI language [BDHH+17] and used as input for a variety of probabilistic model checkers.

4.3.2 Supported DFT elements

We start the comparison between the six DFT semantics by investigating their supported DFT elements. For each DFT element type we check whether the element is supported, which possible variants are supported and whether certain constraints restricting the modelling capabilities must hold.

A summary of the results is given in Table 4.2. We explain the details for each DFT element in the following.

BE | All six semantics support SEs with exponential failure distributions.

Discrete probability distributions such as *Bernoulli* distributions are supported by *Galileo*, *Storm* and the *GSPN framework*. *Galileo* however only supports discrete probability distributions for static subtrees [DVG97]. *Storm* supports such distributions by modelling them with a PDEP_p and a $\mathsf{CONST}_\top$ element which immediately triggers the PDEP_p.

Some of the semantics also support other probability distributions and therefore lift DFT Constraint 7 (Restriction to exponential distributions). *Galileo* also supports

[4] https://www.stormchecker.org/publications/gspn-semantics-for-dfts

Table 4.2: Supported elements in six DFT semantics

		Galileo	*CR-GSPN*	*DFTCalc*	*Storm*	*IOSA*	*GSPN fr.*
BE	exponential	✓	✓	✓	✓	✓	✓
	Bernoulli	(✓)[1]	✗	✗	(✓)[2]	✗	✓
	other	Weibull	✗	ph-ty.	✗	cont.	(ph-ty.)[3]
PAND	type	≤	≤	<	≤	≤[4]	<, ≤
POR	type	✗	✗	✗	≤	✗	<, ≤
SPARE	inputs	BE	BE[5]	gate	gate	BE[6]	gate
	sharing	✓	✗	✓	✓	✓	✓
	claiming	lr	lr	lr	lr	arb.	arb.
	nested claim	✗	✗	late	early	✗	both
	nested fail			early	early		both
FDEP	trigger	gate	gate	gate	gate	(gate)[7]	gate
	dependent	BE	gate	gate	BE	(gate)[7]	BE
PDEP	support	✗	✗	✗	✓	✗	✓
SEQ	type	≤	<	(<)[8]	≤	✗	≤
	inputs	gate	BE	(gate)[8]	gate	✗	BE[6]
MUTEX	support	(✓)[9]	✗	✓	✓	✗	✓

[1] only in static subtrees [2] via PDEP [3] not implemented [4] w.r.t. to time
[5] only one spare child and no cold SE [6] no dependencies to inputs
[7] via OR [8] via cold SPARE (incorrect) [9] via SEQ and AND

Weibull distributions [MDCS98]. *DFTCalc* and the *GSPN framework* support arbitrary *phase-type* distributions as they can be modelled by CTMCs. However, support for phase-type distributions is not yet implemented in the *GSPN framework*. *IOSA* is the most flexible and allows arbitrary continuous probability distributions as it is based on stochastic automata.

Static gates | All six semantics support the static gates AND, OR and VOT_k. We therefore omit these gates from Table 4.2.

Priority gates | The PAND gate is supported by all semantics. Most of the semantics use the *inclusive* variant $\mathrm{PAND}_{\leq}$. Note that in *DFTCalc*, all failures are strictly ordered and no simultaneous failures can occur. Thus, *DFTCalc* uses the *exclusive* variant $\mathrm{PAND}_{<}$. The *GSPN framework* is the only semantics which supports both variants of PAND. Note that for *CR-GSPN*, the entry in [4, Table 1] is incorrect. The PAND in

CR-GSPN is *inclusive* as can be deduced from [Rai05, Fig. 7(b)].

IOSA handles PAND in a unique way: the ordering requirement only considers a timed ordering and not a causal one. The I/O stochastic automaton for a PAND synchronises with all timed transitions to recognize that time has passed. In our concept with an *observed trace order*, we could mimic the *IOSA* behaviour by evaluating PANDs only after all other elements have already been evaluated.

The POR gate is only supported by *Storm* and the *GSPN framework*.

SPARE gate | The SPARE-gate yields the most differences between the six semantics. We start by looking at the supported type of inputs for a SPARE. The original semantics *Galileo* as well as *CR-GSPN* and *IOSA* only allow BEs as children of a SPARE. *CR-GSPN* even further restricts the inputs and only allows exactly one spare child which additionally cannot be a cold SE. *IOSA* does not allow any spare child to also be a child of an FDEP. That way, simultaneous failures of multiple spare children and the occurrence of a *spare race* can be prevented. The *DFTCalc* semantics was the first semantics to allow gates—and therefore complete subtrees—as inputs for a SPARE. Both *Storm* and the *GSPN framework* follow these steps and also allow gates as inputs. The three semantics require that DFT Constraint 8 (Independence of modules) is satisfied. In the *DFTCalc* paper [BCS10] this is referred to as 'activation independence'. Note that in contrast to the entry in [4, Table 2], the primary child of a SPARE cannot be shared for *DFTCalc*: 'The first nondummy input of a SPARE gate (i.e., its primary) cannot be an input to another SPARE gate, i.e., primary components cannot be shared.' [BCS10]

Sharing spare children between different SPARE gates is one of the main benefits of using SPAREs. All semantics except *CR-GSPN* allow sharing. *CR-GSPN* therefore only supports a very limited fraction of the usual functionality of SPARE gates.

All six semantics support claiming in order from left to right. *IOSA* and the *GSPN framework* also allow arbitrary claiming orders and therefore offer more modelling flexibility. *IOSA* however requires the claiming strategy to be deterministic whereas the *GSPN framework* also allows randomized and non-deterministic claiming strategies.

For the three semantics which support arbitrary subtrees as SPARE inputs, we investigate the handling of nested SPAREs. *DFTCalc* uses *late claiming* which only happens after activation of the SPARE. In contrast, it uses *early failing*, i.e. inactive SPAREs can fail even though they cannot claim yet. *Storm* uses *early claiming* and *early failing*, i.e. the activation status of a SPARE has no influence and passive SPAREs behave the same as active ones. The *GSPN framework* supports both variants—early and late—for both claiming and failing.

Dependencies | All six semantics support FDEPs. *IOSA* does not have native support for FDEPs, but instead models them via OR-gates. As seen in Section 4.1.2, this translation is in general incorrect. In this specific instance it is correct though, because in *IOSA*, dependencies cannot trigger spare children and no SEQ is present. Note that due to translating FDEP into OR, *cyclic* dependencies cannot be modelled in *IOSA*.

All six semantics allow gates as the trigger of an FDEP. *Galileo*, *Storm* and the *GSPN framework* require the dependent children to be BEs, i.e. DFT Constraint 11 (Dependent events of PDEPs are BEs) must hold. As seen, *IOSA* uses OR-gates and therefore allows gates as dependent children. *CR-GSPN* and *DFTCalc* explicitly allow gates as children. These three semantics handle the failure of a dependent gate by letting the FDEP output a failure signal to the parents of the dependent gate instead of letting the gate itself output the signal. Thus, the gate itself is technically not yet failed. In all three semantics, the issue with triggered SPAREs as presented in Example 4.2 on page 139 persists. A dependent SPARE gate can still claim and activate children even though it should be failed due to the FDEP. We therefore think that the behaviour of an FDEP in *CR-GSPN*, *DFTCalc* and *IOSA* is incorrect for gates as children.

CR-GSPN is the only semantics where an FDEP itself can also fail. In *CR-GSPN*, an FDEP fails when the trigger fails.

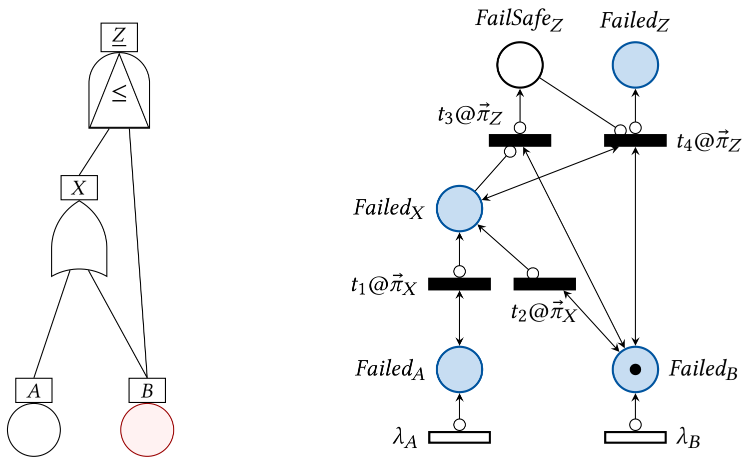
Only *Storm* and the *GSPN framework* support the more general PDEP$_p$.

Restrictors | Lastly, we consider the support for SEQ. *IOSA* does not support any restrictor at all. *DFTCalc* shortly mentions that a SEQ can be modelled by a cold SPARE. However, as already discussed in Section 4.1.3, this translation is in general not correct. The translation is incorrect for *DFTCalc* as well, because none of the problems indicated in Section 4.1.3 are prevented. For the remaining four semantics we consider the supported SEQ variant. *CR-GSPN* supports the *exclusive* variant SEQ$_<$. *Storm* and the *GSPN framework* both support the *inclusive* variant SEQ$_\leq$. For *Galileo*, the question of handling simultaneous failures in a SEQ was posed in [CS98], but it was not indicated how it was resolved. Looking at the corresponding slides[5], we think the supported variant should be an *inclusive* SEQ$_\leq$. As with the FDEP, *CR-GSPN* is the only semantics where an SEQ itself can fail. In *CR-GSPN*, an SEQ fails when the right-most child fails. In other words, the failure behaviour of a SEQ can also be represented by an AND.

CR-GSPN and *GSPN framework* only support BEs as children of SEQ. For *CR-GSPN*, it is not clear from [Rai05, Fig. 7(c)] whether a spare child can also be a child of a SEQ. The *GSPN framework* does not allow children of a SEQ to also be the child of a dependency, i.e. DFT Constraint 14 (No dependencies to inputs of SEQ) must hold. *Galileo* and *Storm* support gates—and therefore arbitrary subtrees—as children of SEQs.

The MUTEX is supported in *Galileo* by a translation to SEQ and AND as presented in Figure 3.40 on page 124. In *DFTCalc*, *Storm* and the *GSPN framework*, MUTEX is natively supported whereas *CR-GSPN* and *IOSA* do not support MUTEX.

[5] http://www.coppit.org/papers/formal_spec_for_tools.pdf, slide 18



(a) DFT $\mathcal{F}_1$ (b) GSPN $\mathcal{G}_{\mathcal{F}_1}$

Figure 4.15: Example for failure propagation

4.4 Semantic issues

In the previous Chapter 3 we investigated the behaviour of each DFT element in detail. It is however not sufficient to consider each gate in isolation. Instead, the interplay between different elements in a complete DFT plays a crucial rule and has to be examined as well. Examples of this interplay were already seen in Sections 4.1 and 4.2. In the following, we present three semantic issues which are introduced through combining DFT elements in a complete DFT. We will present the different ways how to handle failure propagation, failure forwarding through FDEPs and the possible occurrence of non-determinism. For each of the six semantics we investigate how these semantic issues are handled and recapture the results of [4].

Seven semantic issues for DFTs were already identified in [CSD00]. Among them are cyclic dependencies of FDEPs, cascades of FDEPs, *spare races* or BEs which are children of cold and hot SPAREs. All of these issues are addressed and handled in our *GSPN framework*. In particular, we will show in Section 4.4.2 how issues related to dependencies (cf. [CSD00, Issues #2, #5, #6, #7]) are handled in our *GSPN framework*.

4.4.1 Failure propagation

·failure propagation

We start by investigating the different ways how failures can be *propagated* in a DFT. Some examples of these differences were already shown before. We will use the example DFT $\mathcal{F}_1$ in Figure 4.15a in the following. This DFT was previously introduced in Figure 3.1a on page 57. The corresponding (simplified) GSPN $\mathcal{G}_{\mathcal{F}_1}$ is given in Figure 4.15b. The GSPN was previously presented in Figure 3.1c.

We consider the ftrace $\boldsymbol{\tau_f} = \mathbf{f}_B$ where BE B has failed. This is indicated by the red colour for B in Figure 4.15a. In the corresponding GSPN, this corresponds to the marking $m_1 = \mathsf{m}(\mathbf{f}_B)$ where place $Failed_B$ contains a token, i.e. $m_1(Failed_B) = 1$.

The question now is: in which order are failures propagated through the DFT? In the example DFT $\mathcal{F}_1$, either OR X or PAND$_\leq$ Z can be evaluated next. This can also be clearly seen in the corresponding GSPN $\mathcal{G}_{\mathcal{F}_1}$. In marking m_1, both transitions t_2 and t_3 are in concession, i.e. $t_2, t_3 \in \mathsf{conc}(m_1)$. The choice which of these transitions can fire is determined by the priority variables $\vec{\pi}_X$ (for t_2) and $\vec{\pi}_Z$ (for t_3). The enabled transitions correspond to the element(s) in the DFT which should be evaluated next. In total, we distinguish three cases:

$$
\begin{aligned}
\vec{\pi}_X &> \vec{\pi}_Z & t_2 &\in \mathsf{enabled}(m_1) & \mathsf{NextEval}^{Sem_1}(\mathbf{f}_B) &= \{X\}\\
\vec{\pi}_X &< \vec{\pi}_Z & t_3 &\in \mathsf{enabled}(m_1) & \mathsf{NextEval}^{Sem_2}(\mathbf{f}_B) &= \{Z\}\\
\vec{\pi}_X &= \vec{\pi}_Z & t_2, t_3 &\in \mathsf{enabled}(m_1) & \mathsf{NextEval}^{Sem_3}(\mathbf{f}_B) &= \{X, Z\}\,.
\end{aligned}
$$

If for example $\vec{\pi}_X > \vec{\pi}_Z$, then only transition t_2 is enabled. Firing the transition places a token in $Failed_X$ indicating the failure of X. This corresponds to a semantics Sem_1 which first evaluates OR X and yields event f_X. For the second semantics Sem_2, PAND Z is evaluated first. For the third semantics Sem_3, both orders are still possible and will be considered. The resulting traces are as follows:

$$
\begin{aligned}
\tau_1 &= \mathbf{f}_B\, \mathrm{f}_X\, \mathrm{f}_Z\\
\tau_2 &= \mathbf{f}_B \oslash_Z \mathrm{f}_X \oslash_Z.
\end{aligned}
$$

Trace τ_1 corresponds to Sem_1 where X is evaluated next. Trace τ_2 corresponds to Sem_2 where Z is evaluated first. For Sem_3, both traces are considered. In summary, we have:

$$
\begin{aligned}
\mathsf{Traces}^{Sem_1}(\mathbf{f}_B) &= \{\tau_1\}\\
\mathsf{Traces}^{Sem_2}(\mathbf{f}_B) &= \{\tau_2\}\\
\mathsf{Traces}^{Sem_3}(\mathbf{f}_B) &= \{\tau_1, \tau_2\}\,.
\end{aligned}
$$

For Sem_1, the DFT is always failed after $\mathbf{f}_B$ whereas for Sem_2 it is always fail-safe. Thus, the choice of *evaluation order* has drastic consequences on the overall state of the DFT.

Failure propagation in semantics | We evaluate how failure propagation is handled by the six considered semantics from the literature.

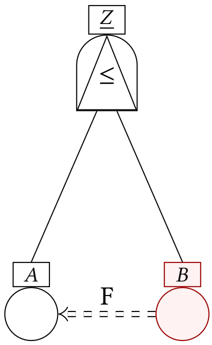
CR-GSPN, *DFTCalc* and *IOSA* consider all possible propagation orders. That means, each element could be evaluated next independent of the given trace. In the GSPN framework that means all priority variables have the same value. In our example, we have $\vec{\pi}_Z = \vec{\pi}_X$. In principle, this leads to non-determinism in the model because all possibilities must be considered and there is no preference for one possibility over the other. We investigate the handling of possible non-determinism in Section 4.4.3. Note that for the given example DFT, *IOSA* would still yield a unique state of the DFT. The PAND of *IOSA* only becomes fail-safe if the right child fails strictly—w.r.t. the time—before the left one. In this example, both B and X fail simultaneously w.r.t. the time and therefore Z always fails. This is modelled in the GSPN framework by evaluating PANDs last, i.e. PANDs get the lowest possible priority. In our example that means $\vec{\pi}_Z < \vec{\pi}_X$. As X is always evaluated before Z, only trace τ_1 is valid for *IOSA*.

The failure propagation for both *Galileo* and *Storm* can be thought of as similar to a fix-point computation. Failures are propagated through the DFT until no element changes its state any more. *Galileo* ensures that no element changes after its evaluation by imposing a fixed evaluation order depending on the type of gate. The order first evaluates SPAREs, then PANDs and evaluates SEQs last [MCSD99]. Note that this fixed order is only correct, because for example SPAREs only have BEs as inputs and therefore cannot change due to evaluations of other gates, such as PANDs. The evaluation order of static gates is not stated in [MCSD99], but we assume it is interleaved with PANDs. We model this fixed order in the GSPN framework by giving all SPAREs a higher priority than PANDs and static gates which in turn have a higher priority than SEQs.

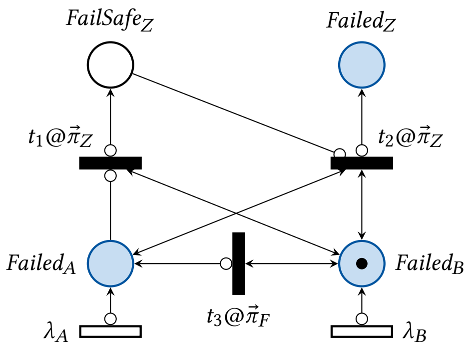
Due to DFT Constraint 1 (DFT is acyclic), the computation of a 'fix-point' can also be performed by evaluating each element only once in a *bottom-up* manner. The crucial point here is that each DFT element is only evaluated after all of its children have been evaluated. That way, the state of the children cannot change after the gate is evaluated. This approach is used in *Storm*. We mimic this behaviour in the GSPN framework by ensuring that the priority of an element is always lower than the priority of all its children. In our example GSPN that means $\vec{\pi}_Z < \vec{\pi}_X$ and transition t_2 always fires first.

The *GSPN framework* can capture all other five semantics by instantiating the priority variables accordingly.

Discussion | Failures instantaneously propagate through a DFT. For a priority gate, the order of these propagations should therefore not be relevant regarding its failure behaviour. We therefore think that trace τ_2 in the example DFT—which leads to a fail-safe PAND—is not desirable. In principle, it is certainly good to consider all possible behaviours of a DFT. However, in case of the failure propagation, we think that traces such as τ_2 have no real-world equivalent and therefore should not be considered. In particular, we think the failure status of gates should only depend on the ftrace of the BEs and not on the order of propagation in the DFT. We therefore think that the bottom-up propagation most accurately reflects the desired behaviour as it resembles the computation of a fix-point. Additionally, bottom-up propagation always yields a



(a) DFT $\mathcal{F}_2$



(b) GSPN $\mathcal{G}_{\mathcal{F}_2}$

Figure 4.16: Example for forwarding in dependencies

unique DFT state. This is similar to static fault trees in which the evaluation order is irrelevant as all orders lead to the same result.

4.4.2 Forwarding in dependencies

We already discussed the propagation of failures through a DFT. A similar case is the *forwarding of failures* through an FDEP. We know that in very limited cases FDEPs can be replaced by ORs (cf. Proposition 4.1 on page 140). One could therefore argue that in principle, failure forwarding should behave the same as failure propagation. As we will see, that is not true for all of the considered semantics. ·failure forwarding

An example DFT $\mathcal{F}_2$ for failure forwarding is given in Figure 4.16a. The DFT contains an inclusive PAND$_{\leq}$ Z and a dependency F which forwards the failure of B to A. The corresponding GSPN $\mathcal{G}_{\mathcal{F}_2}$ is depicted in Figure 4.16b. We consider ftrace $\tau'_{\mathsf{f}} = \mathbf{f}_B$ where B failed. In the GSPN, this corresponds to marking $m_2 = \mathsf{m}(\mathbf{f}_B)$ where place $Failed_B$ contains a token.

The question now is: in which order are failures forwarded by dependencies compared to failure propagation? Similar to the previous example, we again have two possibilities. In $\mathcal{G}_{\mathcal{F}_2}$ both transitions t_1 and t_3 are in concession and could fire. In the DFT $\mathcal{F}_2$, this corresponds to the question whether Z or F are evaluated next. Once again, the choice depends on the semantics and the given priority variables:

$$
\begin{aligned}
&\vec{\pi}_Z > \vec{\pi}_F && t_1 \in \mathsf{enabled}(m_2) && \mathsf{NextEval}^{Sem_1}(\mathbf{f}_B) = \{Z\}\\
&\vec{\pi}_Z < \vec{\pi}_F && t_3 \in \mathsf{enabled}(m_2) && \mathsf{NextEval}^{Sem_2}(\mathbf{f}_B) = \{F\}\\
&\vec{\pi}_Z = \vec{\pi}_F && t_1, t_3 \in \mathsf{enabled}(m_2) && \mathsf{NextEval}^{Sem_3}(\mathbf{f}_B) = \{Z, F\}\,.
\end{aligned}
$$

Choosing Z to be evaluated next leads to trace

$$\tau_1' = \mathbf{f}_B \oslash_Z \mathrm{f}_A \oslash_Z,$$

and choosing F instead leads to trace

$$\tau_2' = \mathbf{f}_B \, \mathrm{f}_A \, \mathrm{f}_Z.$$

Similar to failure propagation, the choice of evaluation order for dependencies can lead to completely different DFT states, e.g. failed or fail-safe PAND$_{\leq}$ in our example.

Failure forwarding in semantics | Most of the considered semantics handle failure forwarding caused by dependencies similar to failure propagation. *CR-GSPN* and *DFTCalc* allow arbitrary orders and FDEPs can be evaluated interleaved with all other DFT elements. *IOSA* replaces FDEPs by ORs and therefore the same considerations as for failure propagation apply.

For *Galileo* the handling of dependencies needs to ensure that the failure of dependent elements is considered to be simultaneous to the failure of the trigger. This can be achieved by always evaluating FDEPs first [MCSD99], i.e. FDEPs get the highest priority in the corresponding GSPN.

Storm places focus on the failure of BEs. After the failure of a BE, the failure is propagated through the DFT before considering the next failure of a BE. That way, it is always ensured that the propagation is finished before the next failure of a BE is considered. For internal failures, only one SE can fail at a time (cf. Remark 2.2 on page 32). However, in case of FDEPs, dependent elements could fail simultaneously which would for example lead to a *spare race*. Moreover, dependent elements could fail before the previous propagation step was finished. *Storm* prevents these scenarios by evaluating FDEPs last, i.e. after all other DFT elements were already evaluated. In the GSPN, this behaviour corresponds to giving FDEPs the lowest priority.

The *GSPN framework* allows to specify the required evaluation orders by instantiating the priority variables accordingly. It therefore captures the other five semantics.

Discussion | Most of the considered semantics agree that dependencies should not be handled differently than other DFT elements. Failure forwarding therefore behaves the same as failure propagation. The exception is *Storm* which considers failure forwarding after the failure propagation. As a result, the translation from FDEPs to ORs—if possible—changes the evaluation order and therefore also the results for *Storm*. One can think of the 'late' evaluation of FDEPs as modelling an (unobservable) delay within the failure forwarding. The usual gates model 'virtual' failure propagation, because the gate itself is not physically present in the real system. Instead, a gate only represents an abstract subsystem by grouping several physical components—

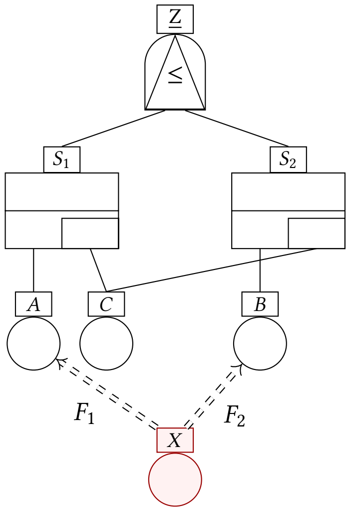


Figure 4.17: Example DFT $\mathcal{F}_3$ for non-determinism (in spare races)

the BEs—together. In contrast, failure forwarding leads to the failure of a BE, i.e. a physical component which exists in the real system. One could therefore argue that this failure forwarding comes with an (unobservable) delay. Note that there exist fault tree extensions allowing explicit modelling of delayed failure propagation [Glu07; KGF07].

We think that in most cases, FDEPs should not behave differently than other gates and therefore should be evaluated either before or interleaved with other gates. Performing failure forwarding after failure propagation should only be used if one wants to explicitly model a difference between these two behaviours, e.g. for modelling 'delays' in failure forwarding. In any case, it is important to explicitly communicate the chosen behaviour to the user.

4.4.3 Non-determinism

We have seen that in some semantics multiple traces are valid for a given ftrace. Moreover, for scenarios such as spare races, several traces might also be possible. We consider the handling of such *non-determinism* in the following. ·non-determinism

An example for a *spare race* is given by the DFT $\mathcal{F}_3$ in Figure 4.17. The corresponding GSPN $\mathcal{G}_{\mathcal{F}_3}$ is depicted in Figure 4.18. Note that we simplified the GSPN by removing some places and transitions which are not necessary in the given scenarios.

In the following, we consider the ftrace $\tau_\mathrm{f}'' = \mathrm{f}_X$ where BE X has failed. In $\mathcal{F}_3$, both

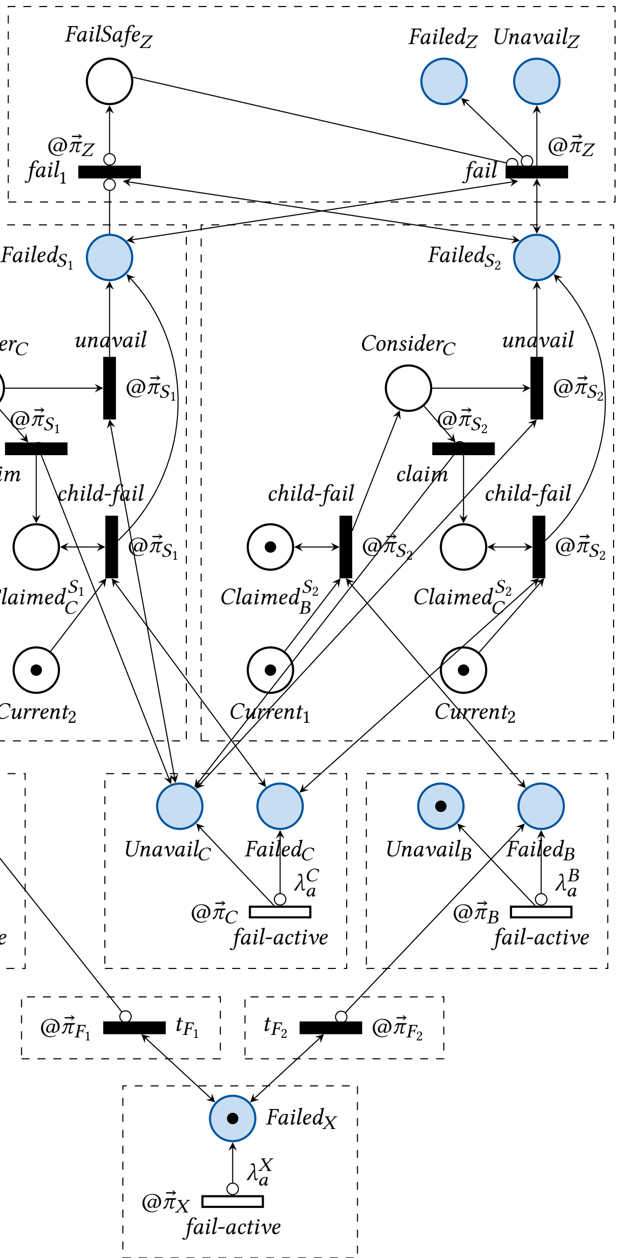


Figure 4.18: GSPN $\mathcal{G}_{\mathcal{F}_3}$ for non-determinism

FDEPs F_1 and F_2 can forward the failure to the BEs A and B. As a result, both SPAREs S_1 and S_2 need to claim a new spare child. The only available spare child is the shared spare C and we have a spare race. This behaviour can also be observed in $\mathcal{G}_{\mathcal{F}_3}$. The corresponding marking is $m_3 = \mathsf{m}(\mathbf{f}_X)$. The failure of X is indicated by a token in Failed_X. As a result, the transitions t_{F_1} and t_{F_2} of both FDEPs are in concession in m_3. In all considered semantics, the priority variables of both FDEPs are equal: $\vec{\pi}_{F_1} = \vec{\pi}_{F_2}$. Therefore, both transitions are enabled and could fire, i.e. there is a *conflict* in m_3. In summary we have

$$t_{F_1}, t_{F_2} \in \mathsf{enabled}(m_3) \qquad\qquad \mathsf{NextEval}^{Sem}(\mathbf{f}_X) = \{F_1, F_2\}\,.$$

The question is now, how are conflicts between multiple evaluation orders resolved?

We distinguish two possibilities depending on the *partitioning* of the GSPN transitions. If both transitions are in separate partitions—i.e. $D, D' \in \mathcal{D}$, $D \neq D'$ and $t_{F_1} \in D$, $t_{F_2} \in D'$—then non-deterministically either t_{F_1} or t_{F_2} is chosen. If both transitions are in the same partitioning—i.e. $t_{F_1}, t_{F_2} \in D \in \mathcal{D}$—then transition t_{F_1} is chosen with probability $\mathbf{P}_D(t_{F_1}, m_3)$ and transition t_{F_2} with probability $\mathbf{P}_D(t_{F_2}, m_3)$. 4

We assume that FDEP F_1 was chosen first. Depending on the order of failure propagation and failure forwarding we get the following traces:

$$\begin{aligned}
\tau_1'' &= \mathbf{f}_X\ \mathsf{f}_A\ \mathsf{cl}_C^{S_1}\ \mathsf{f}_B\ \mathsf{f}_{S_2}\ \oslash_Z \\
\tau_2'' &= \mathbf{f}_X\ \mathsf{f}_A\ \mathsf{f}_B\ \mathsf{cl}_C^{S_1}\ \mathsf{f}_{S_2}\ \oslash_Z \\
\tau_3'' &= \mathbf{f}_X\ \mathsf{f}_A\ \mathsf{f}_B\ \mathsf{cl}_C^{S_2}\ \mathsf{f}_{S_1}\ \oslash_Z\,.
\end{aligned}$$

Note that in a semantics which performs failure forwarding last, only trace τ_1'' can occur. In the semantics with interleaved failure forwarding, all three traces are possible.

We see that τ_2 and τ_3 both contain a conflict after the failures of A and B. The conflict here is between the two SPAREs and therefore constitutes a spare race. In the GSPN, the failures of A and B through transitions t_{F_1} and t_{F_2} yield tokens in Failed_A and Failed_B. Afterwards, both transitions *child-fail* can fire and the two places $\mathit{Consider}_C$ of both S_1 and S_2 contain a token. As a result, the two transitions *claim* of both SPAREs S_1 and S_2 are in conflict. This conflict represents the spare race for spare child C. The one SPARE which 'wins' the spare race lets its transition *claim* fire first. This places a token in $\mathit{Unavail}_C$ and—through the inhibitor arc—disables the other *claim* transition. This conflict between the two SPAREs can be resolved either non-deterministically or probabilistically.

Non-determinism in semantics | The semantics *Galileo* and *CR-GSPN* resolve all conflicts probabilistically and do not use any non-determinism. In the corresponding GSPN, all transitions are part of the same partition, i.e. $\mathcal{D} = \{T_i\}$. In *Galileo* for example, non-determinism can occur for spare races. The non-determinism is resolved by a uniform distribution over the possible choices, called ‘sparing scale factor’ [CSD00].

The semantics *DFTCalc*, *Storm* and *IOSA* all support non-determinism. *DFTCalc* and *IOSA* can contain non-determinism for all transitions. That means, in the corresponding GSPN, each transition forms its own partition, i.e. $\mathcal{D} = \{\{t\} \mid t \in T_i\}$. The resulting models for *IOSA* however are always weakly deterministic [MBD20] and all non-determinism is *spurious*. Note that the proof in [MBD20] excludes SPARE-gates, because the authors could not (yet) show the correctness for DFTs with SPAREs.

Storm allows non-determinism, but restricts its occurrence to FDEPs. Nevertheless, we can assume that each transition is its own partition. Failure forwarding through FDEPs happens only after the failure propagation is finished. At this point, all FDEPs which are triggered can be determined. The choice which FDEP fires first is then resolved non-deterministically. As shown in [Jun15, Theorem 4.30], the restrictions DFT Constraint 8 (Independence of modules), DFT Constraint 9 (No constant failed events in primary modules) and DFT Constraint 10 (No sharing of primary modules) ensure that at most one SPARE is claiming at a time and therefore no spare races can take place in *Storm*.

The *GSPN framework* allows all variants of non-deterministic and probabilistic choice by setting the partitioning—and weights—accordingly. In particular, it allows to capture the other five semantics by setting the partitioning accordingly.

Discussion | We think that non-determinism in general should be supported in a DFT semantics and agree with the motivation by Boudali, Crouzen and Stoelinga:

> ‘If the nondeterminism was not intended, then its presence (which is easily detected) indicates that an error occurred during the model specification. Nondeterminism could also be an inherent characteristic of the system being analyzed, and therefore, should be explicitly modeled.’ ([BCS10])

Without non-determinism, situations such as possible spare races are hidden within the model and no feedback is provided to the user. In contrast, using non-determinism explicitly indicates situations in which multiple evaluation orders are possible. Further investigation of the non-determinism can then show whether the non-determinism is spurious or can indeed lead to different results.

4.5 Summary of DFT semantics

We summarize the findings of the previous Sections 4.3 and 4.4 on the six considered DFT semantics. For an overview of the supported DFT elements, we refer once again

Table 4.3: Differences between DFT semantics

Semantics	Failure propagation	Forwarding depend.	Non-determinism
Galileo	SPARE→PAND→SEQ	first	only probabilistically
CR-GSPN	arbitrary	interleaved	only probabilistically
DFTCalc	arbitrary	interleaved	everywhere
Storm	bottom-up	last	only due to FDEPs
IOSA	arbitrary	interleaved	weakly deterministic
GSPN fr.	user definded	user defined	user defined

to Table 4.2 on page 157. The handling of the three semantic issues from Section 4.4 is summarised in Table 4.3. We give a concise representation of the semantic differences in terms of our GSPN framework in Table 4.4. For each semantics, we define the constraints on the *priority variables* $\vec{\pi}$ and the *partitioning* $\mathcal{D}$. The priority variables define the evaluation order and therefore the handling of failure propagation and failure forwarding. The partitioning defines the handling of non-determinism.

Discussion | We discuss the advantages and disadvantages of the considered DFT semantics. Considering the supported DFT elements, *Storm* clearly offers the most modelling flexibility as it supports for example POR and PDEP. The *GSPN framework* also supports all considered DFT elements and even allows to define the variant of priority gates and SPARE to use. The only drawback of *GSPN framework* is that it restricts the children of SEQs to BEs. Of the remaining semantics, *DFTCalc* is the only one which allows gates under SPAREs. This greatly increases the modelling flexibility and we deem it necessary to have a meaningful DFT semantics. One major issue with *DFTCalc* is that SEQ is not correctly supported. The other three semantics *Galileo*, *CR-GSPN* and *IOSA* only support BEs under SPAREs which is a severe modelling restriction. *CR-GSPN* even further restricts the SPARE to only one spare child and no sharing. *IOSA* does not allow any dependencies to SPARE inputs and also does not support restrictions. The DFTs are therefore rather restricted. The major advantage of *IOSA* is that it supports general continuous distributions for SEs.

While offering different traces for a given ftrace is certainly helpful, allowing all possible traces might over-approximate the behaviour of a system too much. For failure propagation and failure forwarding, we think not all possible traces are relevant. Instead, a bottom-up propagation which reflects the intuitive ‘flow’ through the system is most likely the desired behaviour. This behaviour is given by *Galileo* and *Storm*.

Failure propagation and failure forwarding should be handled similar by a semantics. Thus, *Storm* displays an atypical behaviour as it evaluates dependencies last.

In general, supporting non-determinism is crucial to allow modelling flexibility.

Table 4.4: GSPN differences between DFT semantics

Semantics	Priority variables		Partitioning
Galileo	$\vec{\pi}_v > \vec{\pi}_{v'}$ $\vec{\pi}_v > \vec{\pi}_{v'}$ $\vec{\pi}_v > \vec{\pi}_{v'}$ $\vec{\pi}_v > \vec{\pi}_{v'}$ $\vec{\pi}_v > \vec{\pi}_{v'}$	$\forall v \in \mathsf{FDEP}_{\mathcal{F}}, \forall v' \in \mathsf{SPARE}_{\mathcal{F}}$ $\forall v \in \mathsf{SPARE}_{\mathcal{F}}, \forall v' \in \mathsf{PAND}_{\mathcal{F}}$ $\forall v \in \mathsf{PAND}_{\mathcal{F}}, \forall v' \in \mathsf{SEQ}_{\mathcal{F}}$ $\forall v \in \mathsf{SPARE}_{\mathcal{F}}, \forall v' \in \mathsf{StaticGates}_{\mathcal{F}}$ $\forall v \in \mathsf{StaticGates}_{\mathcal{F}}, \forall v' \in \mathsf{PAND}_{\mathcal{F}}$	$\{T_i\}$
CR-GSPN	$\vec{\pi}_v = \vec{\pi}_{v'}$	$\forall v, v' \in \mathcal{F}$	$\{T_i\}$
DFTCalc	$\vec{\pi}_v = \vec{\pi}_{v'}$	$\forall v, v' \in \mathcal{F}$	$\{\{t\} \mid t \in T_i\}$
Storm	$\vec{\pi}_v < \vec{\pi}_{\sigma(v)_i}$ $\vec{\pi}_v < \vec{\pi}_{v'}$	$\forall v \notin \mathsf{PDEP}_{\mathcal{F}}, \forall 1 \leq i \leq \lvert\sigma(v)\rvert$ $\forall v \in \mathsf{PDEP}_{\mathcal{F}}, \forall v' \notin \mathsf{PDEP}_{\mathcal{F}}$	$\{\{t\} \mid t \in T_i\}$
IOSA	$\vec{\pi}_v = \vec{\pi}_{v'}$ $\vec{\pi}_v < \vec{\pi}_{v'}$	$\forall v, v' \notin \mathsf{PAND}_{\mathcal{F}}$ $\forall v \in \mathsf{PAND}_{\mathcal{F}}, \forall v' \notin \mathsf{PAND}_{\mathcal{F}}$	$\{\{t\} \mid t \in T_i\}$
GSPN framework	user defined		user defined
NewGSPN	$\vec{\pi}_v \leq \vec{\pi}_{\sigma(v)_i}$ $\vec{\pi}_v < \vec{\pi}_{\sigma(v)_i}$ $\vec{\pi}_v \leq \vec{\pi}_{\sigma(v)_1}$ $\vec{\pi}_v \geq \vec{\pi}_{\sigma(v)_i}$	$\forall v \in \mathsf{StaticGates}_{\mathcal{F}}, \forall 1 \leq i \leq \lvert\sigma(v)\rvert$ $\forall v \notin \mathsf{StaticGates}_{\mathcal{F}} \cup \mathsf{PDEP}_{\mathcal{F}},$ $\forall 1 \leq i \leq \lvert\sigma(v)\rvert$ $\forall v \in \mathsf{PDEP}_{\mathcal{F}}$ $\forall v \in \mathsf{PDEP}_{\mathcal{F}}, \forall 2 \leq i \leq \lvert\sigma(v)\rvert$	$\{\{t\} \mid t \in T_i\}$

Non-deterministic behaviour should not be hidden, but provides valuable insights into the modelled system. *DFTCalc*, *Storm* and *IOSA* are the most useful in this regard.

The ideal semantics therefore supports as much elements as *Storm* and the *GSPN framework*, allows non-determinism and uses bottom-up computations for failure propagation and failure forwarding. *Storm* is the closest semantics to these requirements but still lacks the optimal failure forwarding. We therefore propose a new semantics *NewGSPN* which captures all the desired properties in the following.

4.5.1 New GSPN semantics

We define *NewGSPN* in terms of the GSPN framework. All elements supported by the *GSPN framework* are also supported by *NewGSPN*. The new semantics should support non-determinism. In the GSPN framework, all transitions form their own partition:

$$\mathcal{D} = \{\{t\} \mid t \in T_i\}.$$

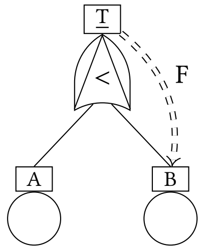


Figure 4.19: Example of downwards FDEP

The failure propagation should be performed in a *bottom-up* manner. That means the priority of all children should be higher than the gate itself:

$$\vec{\pi}_v < \vec{\pi}_{\sigma(v)_i} \quad \forall v \in \mathcal{F}, \forall 1 \leq i \leq |\sigma(v)|.$$

That way, it is ensured that all inputs are evaluated before the gate. For static gates the order of failures is not important. We can therefore lift this restriction and allow that a static gate and its inputs have the same priority:

$$\vec{\pi}_v \leq \vec{\pi}_{\sigma(v)_i} \quad \forall v \in \mathrm{StaticGates}_{\mathcal{F}}, \forall 1 \leq i \leq |\sigma(v)|.$$

Failure forwarding should behave similar to failure propagation. It is therefore natural to impose the same constraints on the priority variables of dependencies. The trigger should always be evaluated before the dependency which should in turn be evaluated before the dependent children:

$$\vec{\pi}_{\sigma(v)_1} \leq \vec{\pi}_v \leq \vec{\pi}_{\sigma(v)_i} \quad \forall v \in \mathrm{PDEP}_{\mathcal{F}}, \forall 2 \leq i \leq |\sigma(v)|.$$

The complete definition of *NewGSPN* is given in the last rows of Table 4.4.

Downward dependencies | Note that the constraints for *NewGSPN* cannot be satisfied for all DFTs. An example of a problematic DFT is given in Figure 4.19. Following the definition of *NewGSPN*, we get the following constraints on the priority variables:

$$\vec{\pi}_A < \vec{\pi}_T, \quad \vec{\pi}_B < \vec{\pi}_T, \quad \vec{\pi}_T \leq \vec{\pi}_F \quad \text{and} \quad \vec{\pi}_F \leq \vec{\pi}_B.$$

Then $\vec{\pi}_B < \vec{\pi}_B$ must hold which is not satisfiable.

The given DFT contains a *downwards dependency*. Even though the DFT is acyclic, it contains a causal cycle. A failure of B is propagated to T and then forwarded to B again. For *NewGSPN*, we have to exclude such scenarios with downwards dependencies. However, this does not restrict the modelling capabilities very much. On the contrary, ·downwards downward dependencies can lead to a paradox. We consider the ftrace $\tau_{\mathrm{f}} = \mathrm{f}_A$ for the example DFT. We assume that the exclusive $\mathrm{POR}_<$ T only considers temporal orders and not causal orders—similar to *IOSA*. A possible trace is $\tau = \mathrm{f}_A\ \mathrm{f}_T\ \mathrm{f}_B\ \oslash_T$. First, $\mathrm{POR}_<$ T fails, because A is failed and B is not. The failure of T leads to the failure of B through FDEP F. Afterwards, we consider T again. Now, A and B are both failed simultaneously—from a temporal perspective. Thus, $\mathrm{POR}_<$ would be fail-safe as the exclusive failure order is violated. However, if $\mathrm{POR}_<$ has not failed, the dependency could not have been triggered in the first place. Downwards dependencies can therefore lead to paradoxical behaviours and it is therefore advisable to exclude them.

NewGSPN is currently only supported in the GSPN framework and no dedicated tool support is available. Thus, *Storm* probably mimics the semantics for DFT analysis the best at the moment.

4.5.2 Other semantics

Apart from the six semantics considered so far, there exist several other DFT semantics. We give a short overview of other DFT semantics in the following. For a thorough summary, we refer to [RS15; JGKS16].

Petri nets | Similar to the translation into GSPNs in [Rai05], there is also a translation of a DFT into a stochastic well-formed net (SWN) [BC04]. The semantics introduces a new element, called ‘replicator’, which models redundant subtrees in the DFT.

Transition systems | In [RB15], DFTs are translated into guarded transition systems. The translation is compositional and each DFT gate is translated into a guarded transition system. Not all elements are natively supported though, FDEPs are modelled via ORs and SEQs as either PANDs or SPAREs. This modelling can however lead to incorrect behaviour, cf. Sections 4.1.2 and 4.1.3.

Bayesian networks | A commonly used approach is to convert a DFT into a Bayesian network (BN) [JN07]. Each DFT element is represented by a random variable in the BN. The random variables corresponding to DFT gates are conditionally dependent on the random variables of the children. There exist translations from DFTs into discrete-time Bayesian networks [BD05b] and continuous-time Bayesian networks [BD06] as well as dynamic Bayesian networks [MPBV+06]. In the first and third approach, time is discretised which leads to inaccuracies but also better analysis performance.

Algebraic encoding | Static fault trees can easily be translated into Boolean algebra. The Boolean algebra can be extended with temporal operators, e.g. ‘before’ or ‘simultaneous’. A DFT element is then represented by a random variables modelling its time of failure. Using temporal operators, dynamic gates such as PAND and SPARE can be encoded in the algebra. Several works used this approach to encode DFTs in an algebra [Mer10; MRL14; EAHT19]. A similar approach uses a sequence algebra to encode DFTs [Rau11].

Trace-based | Lastly, the DFT definition in [Jun15] is based on DFT traces. The failure behaviour of each DFT gate is defined over the traces of component failures. The trace semantics was introduced in order to prove the correctness of the *rewrite rules* given in [Jun15; JGKR+17]. Our definitions of the DFT gates in Chapter 3 are inspired by this trace semantics.

Summary

- Several DFT elements are syntactic sugar:
  - $\mathrm{CONST}_{\perp}$ can be modelled with $\mathrm{PAND}_{\leq}$.
  - VOT-gate subsumes the other static gates AND and OR.
  - FDEP can be represented by PDEP_1.
  - MUTEX can be modelled by SEQ with $\mathrm{CONST}_{\perp}$ and AND as children.
- PAND can be expressed by POR and vice versa. $\mathrm{POR}_{<}$ can express all other priority gates. However, there exist evaluation orders where these constructions are not correct.
- Contrary to common opinion, FDEPs can be translated into ORs only in a very restricted amount of scenarios. In general, the translation is incorrect. FDEPs enlarge expressiveness of DFTs.
- The translation from SEQs into cold SPAREs is also incorrect for most DFTs.
- Nested SPAREs give rise to different claiming variants—early (always) or late (only on activation)—and failing variants (early or late).
- FDEPs triggering the inputs of SEQs can yield 'delayed' failure propagation.
- Existing DFT semantics all support slightly different variants of DFT elements and can significantly differ in the imposed restrictions on DFTs. An extensive summary is given in Table 4.2 on page 157.
- DFT semantics differ in the handling of failure propagation and failure forwarding, and the support for non-determinism. An overview of the semantic choices for all considered semantics is given in Table 4.3 on page 169.
- The presented GSPN framework unifies all considered DFT semantics. The framework can capture each semantics by instantiating the GSPN priority variables accordingly and setting the partitioning in the GSPN. The precise definitions for each DFT semantics are given in Table 4.4 on page 170.

Part III

Analysis

5 State Space Generation

Assumptions

We make the following assumptions in this chapter. The DFT is *conventional*. Additionally, we only consider the inclusive variants $\text{PAND}_{\leq}$, $\text{POR}_{\leq}$ and $\text{SEQ}_{\leq}$. We also only consider binary PDEPs with exactly one dependent child.

After investigating the semantics of dynamic fault trees in detail in Part II, we turn to the analysis of DFTs in the following. *Fault tree analysis (FTA)* uses the (dynamic) fault tree of a real-life system as a model to assess the safety and reliability of the system. The analysis answers questions such as 'how large is the probability that the system has failed within a certain time period?' or 'what is the mean time until a system failure occurs?'. More complex analyses are also possible, for example determining the criticality of single components within the overall system or quantifying how likely the system fails when already degraded.

Our analysis follows in the footsteps of the original approach [DBB92] which uses Markov models as the underlying model of DFTs and lead to the Galileo tool [SDC99]. Using Markov models offers large expressiveness and great flexibility when modelling a system. Additionally, we can exploit a broad range of state-of-the-art algorithms and tools to analyse our systems [HHHK+19; BHKK+21]. The same reasons were already given in the seminal work by Dugan, Bavuso and Boyd:

> 'Markov models are an alternative that is flexible enough to model nearly any such dynamic system. Tools and techniques exist for solving even very large Markov models. However, the construction of a Markov model for any but the simplest system is tedious and error prone.' ([DBB92])

The drawback expressed in the last sentence mostly applies to approaches where a Markov model is created by hand. We will show in this chapter that a fully automatic translation of DFTs into Markov automata is not only possible but very straight-forward in its idea. While this naïve, straight-forward approach already provides the desired results, the generated Markov models can grow quite large. It is thus important to try to keep the state space as small as possible in order to allow for efficient analysis. To this end, we will present several optimisation techniques tailored to DFTs which can lead to drastic state space reductions.

This chapter is based on the publications 'Advancing Dynamic Fault Tree Analysis - Get Succinct State Spaces Fast and Synthesise Failure Rates' [1] and 'Fast Dynamic Fault Tree Analysis by Model Checking Techniques' [6].

5.1 Existing analysis approaches

There exist lots of different analysis approaches for DFTs. An overview of DFT semantics and corresponding analysis approaches was given in Sections 4.3 and 4.5.2. We summarise the main approaches for DFT analysis in the following. For thorough reviews, we refer to [RS15] and [KSYY20].

State-space representation | The semantics considered in Section 4.3 are all based on state spaces, e.g. Markov chains or GSPNs. The tools summarised in Section 4.3.1 use these state spaces for analysis via model-checking techniques or alternative Markov analysis techniques. Examples are the tools Galileo for semantics *Galileo* [SDC99] and DFTCalc for *DFTCalc* [ABBG+13].

A similar approach is done by the tool DRPFTproc [BC04] which translates a DFT into a stochastic well-formed net (SWN)—an extension of Petri nets—and uses existing analysis methods for SWNs.

In [ZMFW09], a DFT is converted into a Petri net and then analysed by simulating runs through the Petri net. The accuracy of the approach depends on the number of runs which are considered. In addition, the used DFT semantics is not clear, because for example, the authors state that the 'Priority-AND gate (PAND) is a special case of SEQ' [ZMFW09] which is not correct, cf. Section 3.8.1.

The compositional approach in [AHMS16] translates each DFT element into a probabilistic automaton (PA). Parallel composition of the PAs yields a Markov decision process (MDP) which is analysed by the probabilistic model checker PRISM [KNP11]. MDPs are a discrete-time model though and therefore this approach cannot faithfully model the passage of time between component failures.

Bayesian networks | For the approaches which translate the DFT into some type of Bayesian network (BN) (cf. Section 4.5.2), dedicated solvers for BNs can be used. Note that the translation into discrete-time BNs [BD05a] or dynamic Bayesian networks (DBNs) [MPB05] needs to discretise time and therefore introduces inaccuracies. Tool support was for example given by Radyban which implements the DFT analysis on DBNs by inference [MPBR08]. However, the tool is not available any more.

Decision diagrams | A prominent approach to solve static fault trees is by using binary decision diagrams (BDD) [Rau93; CM93]. Several extensions of BDDs have been proposed to incorporate the dynamic behaviour of DFTs. In [Rau11], the behaviour of DFTs is encoded as sequences of failures—very similar to our DFT traces. These sequences can be represented in so-called sequence decision diagrams (SDDs) and used for analysis. Similar approaches include translation into sequential binary decision diagrams (SBDDs) [XTD12] or conditional binary decision diagrams (CBDDs) [ZXW20]. In all approaches, the analysis is performed by calculating the probability of all failure sequences encoded in the corresponding BDD. To the best of our knowledge, no tool is publicly available for these approaches. Another approach uses multiple-valued decision diagrams (MDDs) as the underlying representation [Mo14]. In this work, the behaviour under dynamic gates still needs to be modelled by Markov chains. After separate analysis of the dynamic subtrees, a single MDD is created to model the complete system and is subsequently used for analysis.

Minimal cut sequences | Some approaches try to extend minimal cut sets [RS15; TB17] which are heavily used for static fault tree analysis to dynamic fault trees. A common example are minimal cut sequences [TD04; LXZL⁺07]. However, as already stated by Junges et al.: 'Minimal cut sequences are insufficient to characterise the failure behaviour of a DFT.' [JGKS16]

Simulation | There are also approaches based on simulation which do not build a state space but sample the DFT traces instead. Simulation is therefore also applicable to large and complex systems which might be too big for other approaches. The downside is that simulation does not yield exact results, but can only estimate the result with a certain confidence. This can be problematic for safety-critical systems which need strong safety guarantees. The standard simulation approach is to use Monte Carlo simulation for DFT analysis [RGRK⁺09]. However, as the modelled systems are supposed to be as reliable as possible, the occurrence of failures can be very unlikely, i.e. they are rare events. As a result, a large number of simulation runs is necessary to obtain accurate estimations. One approach to tackle this problem is to use rare event simulation [RRBS19; BBMD⁺20] which requires less simulation runs.

Tool support for simulation is provided in DRSIM [RGRK⁺09] and DFTSIM [BNS09], rare event simulation is implemented in DFTRES [BRS20]. The RAATSS tool [MCCD⁺14] translates DFTs into so-called adaptive transition systems and performs analysis via discrete event simulation in MATLAB.

Other tools | Lastly, the DIFTREE tool [MDCS98] combines various solution techniques—Markov chain analysis, BDDs, Monte Carlo simulation—into one framework. Commercial tools for fault tree analysis are also available. Prominent examples

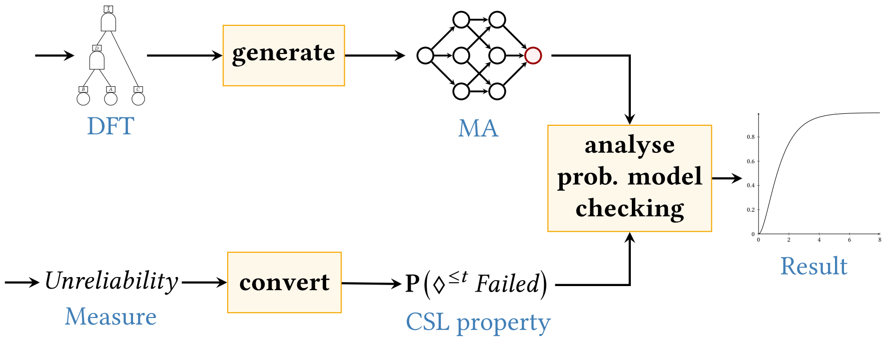


Figure 5.1: Overview of workflow for DFT analysis

are FaultTree+[1] and Windchill FTA[2] which both support (some) dynamic gates.

5.2 Markov chain generation

Our state-space generation follows the same idea as the approach for Galileo presented in [MCSD99]. In contrast to *CTMCs* as used in Galileo, we employ *Markov automata* as the underlying model, because it allows to incorporate non-determinism.

Overview of workflow | The general overview of the analysis workflow is depicted in Figure 5.1. We start with a DFT and a measure as input. From the DFT, we generate a Markov automaton. The details of the generation will be given in this section. The measure we are interested in is expressed as an property in *continuous stochastic logic (CSL)* [BHHK03]. The conversion will be explained in Section 5.3. Finally, the Markov model and the CSL property are used as input for the analysis. The analysis uses existing, state-of-the-art algorithms for probabilistic model checking [GHHK+14; BHKK+21] and returns the desired result.

Generation idea | We start by describing the generation of the Markov automaton. Each *state* of the DFT corresponds to a state in the Markov automaton. Failures of BEs in the DFT are represented by *Markovian transitions* in the MA. The choice which dependency in the DFT triggers next is translated to a non-deterministic choice in the MA with *probabilistic transitions*. The failure of the top-level event of the DFT is

[1] https://www.isograph.com/software/reliability-workbench/
[2] https://www.ptc.com/en/technologies/plm/quality-management

indicated by the *labelling* in the MA. These labels can later be used when formalizing the properties to check.

5.2.1 Mapping of states

We first define the state of a DFT. We use the definitions in Section 2.2.5 on page 40.

Definition 5.1 (State of DFT) Let $\mathcal{F}$ be a DFT and $\tau \in \mathsf{Traces}$ be a trace. The *state* of the DFT after trace τ is denoted by $\mathsf{state}(\tau)$ and is given by ·state

$$\mathsf{state} : \mathsf{Traces} \to \bigtimes_{v \in V \setminus \mathsf{SPARE}_{\mathcal{F}}} \{\mathsf{F}^v, \mathsf{O}^v, \mathsf{FS}^v\} \times \bigtimes_{v \in \mathsf{SPARE}_{\mathcal{F}}} \{\mathsf{F}^v\} \cup \left\{\mathsf{Cl}^v_{v'} \mid v' \in \sigma(v)\right\}.$$

Each DFT element v can either be *operational* (O^v), *failed* (F^v) or *fail-safe* (FS^v). Each SPARE additionally stores which child v' it currently claims ($\mathsf{Cl}^v_{v'}$). More formally, the state $\mathsf{state}(\tau)_v$ of each DFT element v after trace τ is given by:

- For $v \in V \setminus \mathsf{SPARE}_{\mathcal{F}}$:

$$\mathsf{state}(\tau)_v = \begin{cases} \mathsf{F}^v & \text{if } v \in \mathsf{Failed}(\tau), \\ \mathsf{FS}^v & \text{if } v \notin \mathsf{Failed}(\tau') \text{ for all } \tau \subseteq \tau', \\ \mathsf{O}^v & \text{otherwise.} \end{cases}$$

- For $v \in \mathsf{SPARE}_{\mathcal{F}}$:

$$\mathsf{state}(\tau)_v = \begin{cases} \mathsf{F}^v & \text{if } v \in \mathsf{Failed}(\tau), \\ \mathsf{Cl}^v_{v'} & \text{otherwise,} \end{cases}$$

  where $v' = \operatorname{argmax}_{v' \in \sigma(v)} \left\{ i \mid \tau_i = \mathsf{cl}^v_{v'} \right\}$ is the child which was claimed last by v.

The tuple $\mathsf{state}(\tau)$ suffices to uniquely identify the state of a DFT. All other information such as the *active elements* $\mathsf{Active}(\tau)$ can be deduced from this tuple. Note that a SPARE can never be operational without having claimed a child, because *Storm* uses *early claiming* and *early failing* for *nested SPAREs*.

Fail-safe element | An element v is *fail-safe* (FS^v) if it can never fail any more for any trace suffixes. However, in practice, we do not want to consider all possible futures and need a more explicit definition to determine whether an element is fail-safe or not. For example, a $\mathsf{PAND}_{\leq}$ becomes fail-safe if its failure condition (cf. Definition 3.8 on page 75) can never be satisfied any more.

5

fail-safe

Definition 5.2 (Fail-safe element) A $\mathsf{PAND}_{\leq}$ or $\mathsf{POR}_{\leq}$ v is *fail-safe* after trace τ in a DFT $\mathcal{F}$, i.e. $\mathrm{state}(\tau)_v = \mathsf{FS}^v$, if the following condition holds:

$$\exists \sigma(v)_{i-1}, \sigma(v)_i \in \sigma(v) : \mathrm{state}(\tau)_{\sigma(v)_{i-1}} = \mathsf{O}^v \wedge \mathrm{state}(\tau)_{\sigma(v)_i} = \mathsf{F}^v.$$

All gates are fail-safe if enough of their children are fail-safe such that the failure condition cannot be satisfied any more. More formally, $\mathrm{state}(\tau)_v = \mathsf{FS}^v$ for gate v and trace τ if the following condition holds:

- For $v \in \mathsf{AND} \cup \mathsf{PAND}_{\leq}$:

$$\exists v' \in \sigma(v) : \mathrm{state}(\tau)_{v'} = \mathsf{FS}^{v'}.$$

- For $v \in \mathsf{OR}$:

$$\forall v' \in \sigma(v) : \mathrm{state}(\tau)_{v'} = \mathsf{FS}^{v'}.$$

- For $v \in \mathsf{POR}_{\leq}$:

$$\mathrm{state}(\tau)_{\sigma(v)_1} = \mathsf{FS}^{\sigma(v)_1}.$$

- For $v \in \mathsf{VOT}_k$ with $n = |\sigma(v)|$:

$$\exists \{i_1, \ldots, i_{n-k+1}\} \subseteq \{1, \ldots, n\} : \bigwedge_{j=1}^{n-k+1} \mathrm{state}(\tau)_{\sigma(v)_{i_j}} = \mathsf{FS}^{\sigma(v)_{i_j}}.$$

- All other elements—BE, SPARE, PDEP and SEQ—are never fail-safe.

Example 5.1 (DFT state) We use the DFT $\mathcal{F}$ in Figure 5.2 as a running example for the state-space generation. Initially, all elements in the DFT are operational and SPARE S has claimed its primary child C. The corresponding trace is

$$\tau = \uparrow_T \uparrow_P \uparrow_S \uparrow_E \uparrow_R \uparrow_F \uparrow_A \uparrow_B \mathrm{cl}^S_C \uparrow_C.$$

The corresponding state of the DFT is:

$$\mathrm{state}(\tau) = \langle \mathsf{O}^A, \mathsf{O}^B, \mathsf{O}^C, \mathsf{O}^D, \mathsf{O}^E, \mathsf{O}^P, \mathsf{Cl}^S_C, \mathsf{O}^T, \mathsf{O}^F, \mathsf{O}^R \rangle.$$

After the failure of B and the subsequent (successful) triggering of C through $\mathsf{PDEP}_{0.7}$ F, we get the trace

$$\tau' = \tau\, \mathbf{f}_B \oslash_E \oslash_P \mathrm{f}_C\, \mathrm{cl}^S_D \uparrow_D.$$

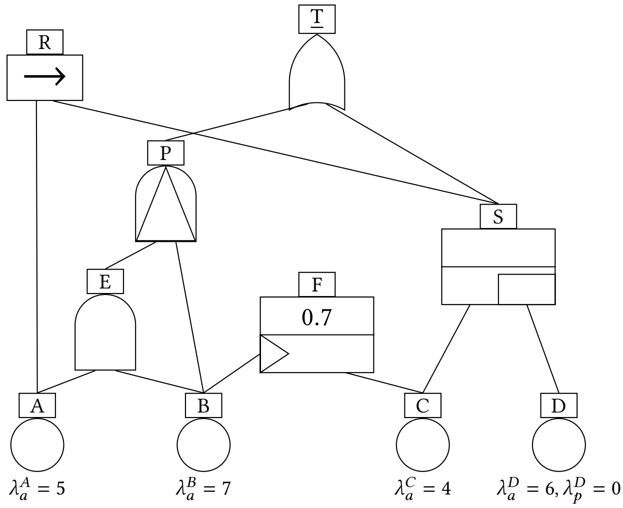


Figure 5.2: Example DFT $\mathcal{F}$

The corresponding state of the DFT is:

$$\text{state}(\tau') = \langle \mathsf{O}^A, \mathsf{F}^B, \mathsf{F}^C, \mathsf{O}^D, \mathsf{O}^E, \mathsf{FS}^P, \mathsf{Cl}^S_D, \mathsf{O}^T, \mathsf{F}^F, \mathsf{O}^R \rangle.$$

PAND P has become fail-safe, because its second child B is failed but the first child E is not. SPARE S has claimed the spare child D. Note that we use F^F to indicate that the PDEP$_{0.7}$ forwarded the failure.

SEQ and PDEP can never fail. We abuse the original notation and let the 'failure' of these elements encode additional information. For SEQ v, we use F^v to mean that the restrictor was violated. This way, we can easily check the state of a SEQ to see whether the current trace is invalid. Similarly, we use F^v for PDEP v to mean that the PDEP performed a coin flip—irrespective of whether the failure was forwarded or not. This way the state of a PDEP changes after a coin-flip and we avoid multiple coin-flips.

5.2.2 State space exploration

We can translate each trace $\tau \in \text{Traces}$ in a DFT into a corresponding state state(τ) in the Markov automaton. A naïve approach therefore would translate a DFT into an MA by creating states for each trace. However, this approach certainly leads to a severe state-space explosion, because we create a state after each evaluation of a DFT element. That means for the failure of each BE we most likely have several subsequent

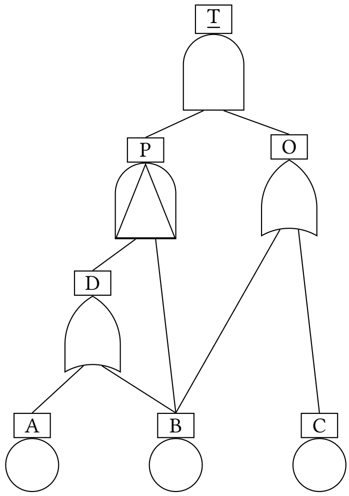


Figure 5.3: Example DFT for bottom-up evaluation order

evaluations and therefore several new states. Thus, this naïve approach is not practical.

A better solution is to exploit the idea that the failure propagation leads to a unique state in the end. In other words, elements can be evaluated in different orders, but the end result will always be the same. Note that this is not true for all possible evaluation orders (cf. Section 4.4.1), but is true for our semantics *Storm* which evaluates elements in a bottom-up order.

Example 5.2 (Bottom-up evaluation) We consider the DFT given in Figure 5.3 and the ftrace $\boldsymbol{\tau_{\mathsf{f}}} = \mathbf{f}_B$.
The corresponding traces for a bottom-up evaluation order—as done in *Storm*—are:

$$\tau_1 = \mathbf{f}_B\, \mathsf{f}_D\, \mathsf{f}_P\, \mathsf{f}_O\, \mathsf{f}_T \in \mathsf{Traces}^{Storm}(\boldsymbol{\tau_{\mathsf{f}}})$$
$$\tau_2 = \mathbf{f}_B\, \mathsf{f}_D\, \mathsf{f}_O\, \mathsf{f}_P\, \mathsf{f}_T \in \mathsf{Traces}^{Storm}(\boldsymbol{\tau_{\mathsf{f}}})$$
$$\tau_3 = \mathbf{f}_B\, \mathsf{f}_O\, \mathsf{f}_D\, \mathsf{f}_P\, \mathsf{f}_T \in \mathsf{Traces}^{Storm}(\boldsymbol{\tau_{\mathsf{f}}})$$

The end result is the same for all three traces:

$$\mathsf{state}(\tau_1) = \mathsf{state}(\tau_2) = \mathsf{state}(\tau_3) = \langle \mathsf{O}^A, \mathsf{F}^B, \mathsf{O}^C, \mathsf{F}^D, \mathsf{F}^P, \mathsf{F}^O, \mathsf{F}^T \rangle.$$

We perform a bottom-up propagation which ignores dependencies. If a BE fails, this failure is propagated through all gates—except dependencies—such that a gate is only evaluated after all its children have been evaluated before. We reflect this propagation (without considering dependencies) by modifying the original *propagation* function given in Definition 2.29 on page 37.

Definition 5.3 (Propagation without dependencies) The next event(s) to evaluate—while ignoring dependencies—are denoted by

$$\mathsf{NextEvalDep}(\tau) = \mathsf{NextEval}^{\mathit{Storm}}(\tau) \setminus \mathsf{PDEP}_{\mathcal{F}}.$$

The *propagation without dependencies* in a DFT is recursively defined as:

·propagation without dependencies

$$\mathsf{PropDep}(\tau) = \begin{cases} \bigcup_{v \in \mathsf{NextEvalDep}(\tau)} \mathsf{PropDep}(\tau\ \mathsf{Eval}(\tau, v)) & \text{if } \mathsf{NextEvalDep}(\tau) \neq \emptyset \\ \{\tau\} & \text{otherwise.} \end{cases}$$

As the DFT is acyclic (cf. DFT Constraint 1 (DFT is acyclic)) and we do not consider dependencies in this propagation—which could introduce cyclic dependencies—there cannot exist any cycle in the propagation. In a bottom-up propagation, if we evaluate a gate, its state cannot change after its evaluation any more, because the state of the children can also not change later on. As a result, the bottom-up propagation—without dependencies—always yields a unique state.

5

Theorem 5.1 (Bottom-up propagation yields unique state) Let $\mathcal{F}$ be a DFT and $\boldsymbol{\tau}_{\mathsf{f}} = \boldsymbol{\tau}_{\mathsf{f}}'\ \mathbf{f}_x$ be an ftrace. Let $\tau \in \mathsf{Traces}^{\mathit{Storm}}(\boldsymbol{\tau}_{\mathsf{f}}')$ be the corresponding trace prefix. Then the propagation without dependencies after $\mathbf{f}_x$ yields a unique state:

$$\text{For all } \tau_1, \tau_2 \in \mathsf{PropDep}(\tau\ \mathbf{f}_x) : \mathsf{state}(\tau_1) = \mathsf{state}(\tau_2).$$

Proof. We prove the statement using the GSPN translation from Chapter 3 on page 55. Let $\vec{\pi}_{\mathsf{PDEP}}$ be the highest priority of a dependency. We start from the marking $m = \mathsf{m}(\tau)$ in the GSPN $\mathcal{G}_{\mathcal{F}}$ which corresponds to trace τ in the DFT. We need to show that all firing sequences starting in m and only using transitions t with priority $\Pi(t) > \vec{\pi}_{\mathsf{PDEP}}$ will end in a unique marking:

$$\text{For all firing sequences } m \xrightarrow{t_1^1} \dots \xrightarrow{t_1^n} m_1^n \text{ and } m \xrightarrow{t_2^1} \dots \xrightarrow{t_2^m} m_2^m \text{ it is } m_1^n = m_2^m.$$

We use two observations for the correctness proof.

1. Firing a transition t in a GSPN template cannot enable transitions with higher priority than $\Pi(t)$. Transitions within a GSPN template have the same priority. Transitions of a template can also be enabled if interface places of the children contain tokens. However, following the definition of the priority variables for *Storm* (cf. Table 4.4 on page 170), children of a gate always have higher

priority than the gate itself, i.e. $\vec{\pi}_v < \vec{\pi}_{\sigma(v)_i}$. The only exception to this rule are PDEP-gates which are explicitly excluded here. Thus, no transitions with higher priority can be enabled.

2. Templates with the same priority are independent of each other, i.e. the order in which transitions of different templates fire is irrelevant. In other words, transitions t_1 and t_2 of different templates can fire in arbitrary order without changing the resulting marking in the end:

$$m \xrightarrow{t_1} m_1 \xrightarrow{t_2} m_2 \text{ and } m \xrightarrow{t_2} m_1' \xrightarrow{t_1} m_2' \text{ with } m_2 = m_2'.$$

Following again the definition of the priority variables, templates with the same priority cannot be children or parents of each other. Thus, firing a transition in one template does not affect the other template at all. Additionally, firing a transition can also not enable transitions in a parent template, because these transition have a lower priority.

In summary, all firing sequences during the failure propagation have the form:

$$m \xrightarrow{t_x} m' \underbrace{\xrightarrow{t_1^1} m_1^1 \xrightarrow{t_1^2} \dots \xrightarrow{t_1^{k_1}}}_{\Pi(t_1^1)} m_1 \underbrace{\xrightarrow{t_2^1} \dots \xrightarrow{t_2^{k_2}}}_{\Pi(t_2^1)} m_2 \dots \underbrace{\xrightarrow{t_n^1} \dots \xrightarrow{t_n^{k_n}}}_{\Pi(t_n^1)} m_n,$$

$$\underbrace{t_1^1 t_1^2 \dots t_1^{k_1}}_{\Pi(t_1^1)} \underbrace{t_2^1 \dots t_2^{k_2}}_{\Pi(t_2^1)} \dots \underbrace{t_n^1 \dots t_n^{k_n}}_{\Pi(t_n^1)},$$

where $\Pi\left(t_1^1\right) > \Pi\left(t_2^1\right) > \cdots > \Pi\left(t_n^1\right)$. This follows from the first observation. Following the second observation, the firing order of transitions $t_1^i, \dots t_1^{k_i}$ can be arbitrary for all i but always yields the same marking m_i. Thus, the marking in the end is always the same for all firing sequences. ■

Using the fact that the bottom-up propagation always yields a unique state, we can improve the naïve state-space generation. Instead of creating a new state after each evaluation step, we only create new states after the complete propagation PropDep(τ) is finished. That way, we omit intermediate states and drastically reduce the resulting state space. Each state in the MA therefore represents the state of the DFT after the failure propagation is completely finished. Each propagation step starts with the failure of exactly one BE and propagates the failure of this BE. Because we ignore dependencies in this propagation step, the BE is also the only BE which can fail during the propagation. That means the result after a propagation is uniquely determined by the BE which fails in the beginning. For each state in the MA, we consider all possible

BE failures and create corresponding transitions for each such failure. The successor state for a transition is then given by the state after propagating the failure of this BE. Following this approach yields the Markov automaton for a DFT.

Definition 5.4 (Markov automaton for DFT) Let $\mathcal{F}$ be a DFT.
The *Markov automaton for DFT $\mathcal{F}$* is given by $\mathcal{M}_{\mathcal{F}} = (S, s_0, \mathsf{Act}, \hookrightarrow, \dashrightarrow, \mathsf{AP}, L)$ with: Markov automaton for DFT

- $S \subseteq \{\mathsf{state}(\tau) \mid \tau \in \mathsf{Traces}\}$,
- $s_0 = \mathsf{state}(\mathsf{PropDep}(\varepsilon))$,
- $\mathsf{Act} = \{v \in \mathsf{PDEP}_{\mathcal{F}}\}$,
- $\hookrightarrow \subseteq S \times \mathsf{Act} \times \mathsf{Distr}(S)$ is defined according to the rule:

$$\frac{v \in \mathsf{PDEP}_p \quad \mathsf{state}(\tau)_v = \mathsf{O}^v \quad \mathsf{state}(\tau)_{\sigma(v)_1} = \mathsf{F}^{\sigma(v)_1} \quad \mathsf{state}(\tau)_{\sigma(v)_2} = \mathsf{O}^{\sigma(v)_2}}{\mathsf{state}(\tau) \overset{v}{\hookrightarrow} \mu}$$

  with distribution μ given by

$$\mu\big(\mathsf{state}(\mathsf{PropDep}(\tau\, \mathsf{f}_{\sigma(v)_2}))\big) = p$$
$$\mu\big(\mathsf{state}(\mathsf{PropDep}(\tau\, \oslash_v))\big) = 1 - p$$

- $\dashrightarrow \subseteq S \times \mathbb{R}_{>0} \times S$ is defined according to the rule:

$$\frac{v \in \mathsf{SE} \quad \mathsf{state}(\tau)_v = \mathsf{O}^v \quad \neg\mathsf{NextDep}(\tau)}{\mathsf{state}(\tau) \overset{\lambda}{\dashrightarrow} \mathsf{state}(\mathsf{PropDep}(\tau\, \mathbf{f}_v))}$$

  with

$$\lambda = \begin{cases} \lambda_a^v & \text{if } v \in \mathsf{Active}(\tau) \\ \lambda_p^v & \text{otherwise,} \end{cases}$$

  and

$$\mathsf{NextDep}(\tau) = \exists v \in \mathsf{PDEP} : \mathsf{state}(\tau)_v = \mathsf{O}^v \wedge \mathsf{state}(\tau)_{\sigma(v)_1} = \mathsf{F}^{\sigma(v)_1}$$
$$\wedge\, \mathsf{state}(\tau)_{\sigma(v)_2} = \mathsf{O}^{\sigma(v)_2}.$$

- $\mathsf{AP} = \{\mathit{Failed}_v \mid v \in \mathcal{F}\} \cup \{\mathit{Failed}\}$,
- $L(\mathsf{state}(\tau)) = \{\mathit{Failed}_v \mid \mathsf{state}(\tau)_v = \mathsf{F}^v\} \cup \begin{cases} \{\mathit{Failed}\} & \text{if } \mathsf{state}(\tau)_{top} = \mathsf{F}^{top} \\ \emptyset & \text{otherwise} \end{cases}.$

We start the state-space generation in the initial state s_0 resulting from the first propagation $\mathrm{PropDep}(\varepsilon)$. This propagation ensures that all SPAREs have claimed their primary child and activation of the elements was performed. From there on, we explore successor states in two ways: by failure forwarding of dependencies or SE failures.

1. Dependencies: If a dependency can fire in a state, we first consider this dependency. A dependency can fire if $\mathrm{NextDep}(\tau)$ is true, i.e. there is an operational dependency ($\mathrm{state}(\tau)_v = \mathrm{O}^v$) whose trigger is failed ($\mathrm{state}(\tau)_{\sigma(v)_1} = \mathrm{F}^{\sigma(v)_1}$) but the dependent child is still operational ($\mathrm{state}(\tau)_{\sigma(v)_2} = \mathrm{O}^{\sigma(v)_2}$). For each dependency v which can fire, we add a *probabilistic transition* with an action corresponding to the dependency. The transition goes from the current state $\mathrm{state}(\tau)$ to state s' where the dependent element $\sigma(v)_2$ has failed, i.e. $s' = \mathrm{state}\big(\mathrm{PropDep}(\tau\, \mathrm{f}_{\sigma(v)_2})\big)$. If the considered dependency is an FDEP, the probability distribution μ goes to s' with probability 1. Otherwise—for a PDEP_p—the probability to go to s' is p and with probability $1-p$ we go to a successor state s'' where the dependent element is not failed, i.e. $s'' = \mathrm{state}(\mathrm{PropDep}(\tau \oslash_v))$. If multiple dependencies can fire in state $\mathrm{state}(\tau)$, we introduce multiple actions which yield non-determinism.
2. SE failures: If no dependency can fire, we consider all possible SE failures. Each SE v which is operational introduces a *Markovian transition* according to its active or passive *failure rate* which goes to successor state $s' = \mathrm{state}(\mathrm{PropDep}(\tau\, \mathbf{f}_v))$.

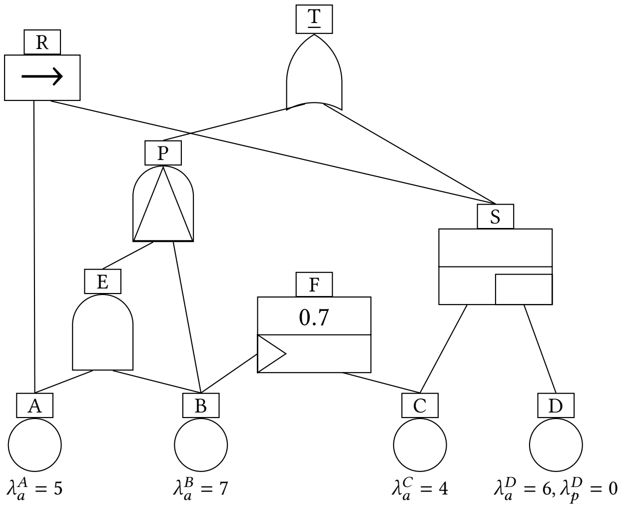
Lastly, we label each state $\mathrm{state}(\tau)$ with those DFT elements which are failed in this state. We introduce a special atomic proposition *Failed* to indicate the failure of the TLE. Note that it often suffices to restrict the labelling to *Failed* as failures of the other DFT elements are not relevant.

Example 5.3 (MA for DFT) We continue with Example 5.1 and again consider the DFT $\mathcal{F}$ restated in Figure 5.4a. The corresponding MA $\mathcal{M}_{\mathcal{F}}$ is (partially) depicted in Figure 5.4b.
We start with the initial state s_0 where all elements are operational. The initial state corresponds to the trace

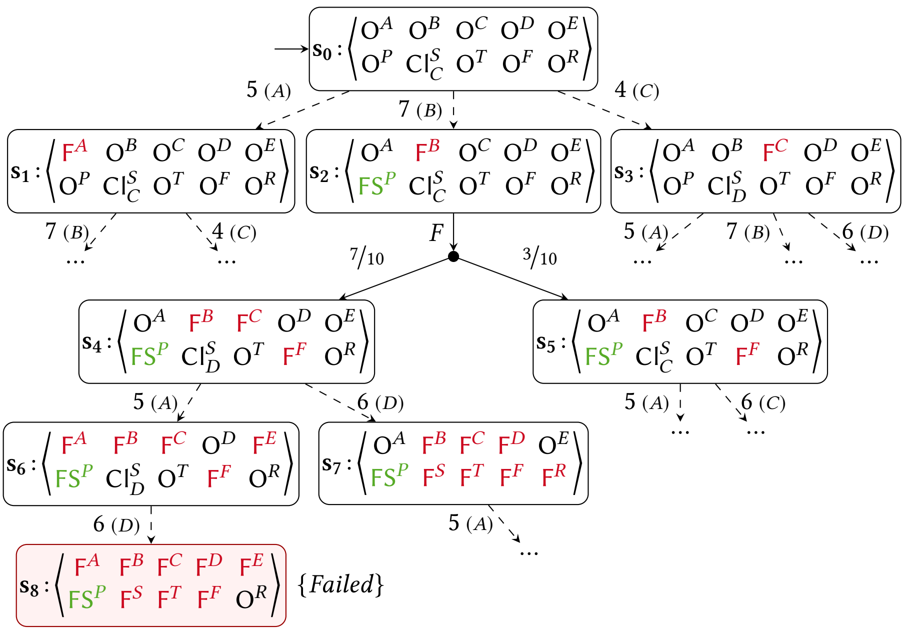
$$\tau_0 = \uparrow_T \uparrow_P \uparrow_S \uparrow_E \uparrow_R \uparrow_F \uparrow_A \uparrow_B \mathrm{cl}_C^S \uparrow_C,$$

i.e. $s_0 = \mathrm{state}(\tau_0)$. In s_0, no dependency can fire and we therefore consider all SE failures. There are three SEs which can fail: A, B and C. D cannot fail, because it is a *cold* BE. The failure of A yields the successor state s_1 which is reached with a Markovian transition and rate $\lambda_a^A = 5$. The failures of B and C yield states s_2 and s_3, respectively.
We consider state s_2 in detail. The state corresponds to trace

$$\tau_2 = \tau_0\ \mathbf{f}_B \oslash_E \oslash_P.$$



(a) Example DFT $\mathcal{F}$ (restated from Figure 5.2 on page 183)



(b) MA $\mathcal{M}_{\mathcal{F}}$

Figure 5.4: Example for state-space generation

5

In state $s_2 = \text{state}(\tau_2)$, PAND P is fail-safe and also B, the trigger of PDEP$_{0.7}$ F, is failed. That means, we first consider this dependency before handling SE failures. The dependency introduces a new outgoing probabilistic transition from s_2 with action F. With probability 0.7 the PDEP$_{0.7}$ successfully forwards the failure and yields state s_4 where the dependent child C is failed. This corresponds to trace

$$\tau_4 = \tau_2 \, \mathrm{f}_C \, \mathrm{cl}_D^S \uparrow_D.$$

With probability 0.3 the failure forwarding of PDEP$_{0.7}$ F is not successful and C stays operational. This yields state s_5 where only F is marked as failed to indicate that the PDEP$_{0.7}$ was already evaluated. Marking the dependency as failed is important, because otherwise the successor state would be s_2 again. We would obtain a self-loop and the coin-flip could be performed until it is successful.
We continue with state s_4 where both B and C are failed and SPARE S has claimed its spare child D. That means, either A or D can fail in this state. The failure of D yields state s_7. It is important to notice here that SPARE S is failed in s_7 but A is not. As a result, SEQ R is violated and this is indicated by setting R to failed. State s_7 is therefore invalid as it corresponds to the invalid trace

$$\tau_7 = \tau_4 \, \mathbf{f}_D \, \mathrm{f}_S \, \mathrm{f}_T \, \lightning_R.$$

The other successor state s_6 is valid and corresponds to trace

$$\tau_6 = \tau_4 \, \mathbf{f}_A \, \mathrm{f}_E \oslash_P \oslash_R.$$

From there, only SE D can fail and yields state s_8 where all elements—except P and R—are failed. The corresponding trace is

$$\tau_8 = \tau_6 \, \mathbf{f}_D \, \mathrm{f}_S \, \mathrm{f}_T \oslash_R.$$

In particular, the TLE T is failed in s_8 and we therefore label this state with *Failed*.

Each state in the MA is created from its predecessor by letting one BE fail—either on its own or through a dependency—and propagating this failure. We can see this for example in the complete trace for state s_8 of Example 5.3:

$$\tau_8 = \underbrace{\uparrow_T \uparrow_P \uparrow_S \uparrow_E \uparrow_R \uparrow_F \uparrow_A \uparrow_B \mathrm{cl}_C^S \uparrow_C}_{s_0} \underbrace{\mathbf{f}_B \oslash_E \oslash_P}_{s_2} \underbrace{\mathrm{f}_C \, \mathrm{cl}_D^S \uparrow_D}_{s_4} \underbrace{\mathbf{f}_A \, \mathrm{f}_E \oslash_P \oslash_R}_{s_6} \underbrace{\mathbf{f}_D \, \mathrm{f}_S \, \mathrm{f}_T \oslash_R}_{s_8}.$$

Each state in the MA can therefore be uniquely identified by the corresponding sequence of BE failures. In the following, we will often use a more compact representation by restricting the traces to the BE failures. For trace τ_8, the compact trace would be:

$$\tau_8' = \mathbf{f}_B\, \mathrm{f}_C\, \mathbf{f}_A\, \mathbf{f}_D.$$

Note that this trace exactly corresponds to the sequence of transitions required to reach state s_8 in Figure 5.4b. Comparing τ_8 and τ_8', we can also see that instead of creating 22 individual states for each evaluation step, we only need five states: one for the initialization and four for the subsequent BE failures.

5.2.2.1 Properties of resulting MA

We give some properties of the resulting MA $\mathcal{M}_\mathcal{F}$ for an arbitrary DFT $\mathcal{F}$.

First of all, the state-space generation terminates.

Lemma 5.2 Let $\mathcal{F}$ be a DFT. The MA $\mathcal{M}_\mathcal{F}$ according to Definition 5.4 is finite.

Proof (sketch). We have seen that each state s in $\mathcal{M}_\mathcal{F}$ corresponds to a compact trace τ' consisting only of BE failures. The number of BEs in $\mathcal{F}$ is finite. Thus, the number of BE traces is finite and the number of states is also finite. ■

The resulting MA contains no loops.

Lemma 5.3 (MA for DFT is acyclic) Let $\mathcal{F}$ be a DFT. The MA $\mathcal{M}_\mathcal{F}$ is *acyclic*.

Proof. Let $k_s = |\{v \in \mathcal{F} \mid s_v = \mathsf{O}^v\}|$ be the number of operational elements in state $s \in \mathcal{M}_\mathcal{F}$. For each Markovian transition $s \overset{\lambda}{\dashrightarrow} s'$ in $\mathcal{M}_\mathcal{F}$, we have $k_s > k_{s'}$, because at least the SE corresponding to the failure rate λ has failed. For each probabilistic transition $s \overset{F_v}{\hookrightarrow} \mu$ and successor state s' with $\mu(s') > 0$, we also have $k_s > k_{s'}$, because at least the corresponding PDEP F_v is set to failed F^v. Therefore, for all transitions in the MA, the number of operational elements in a successor state is strictly smaller than the number in the original state. Thus, no cycles are possible, because otherwise the number of operational elements would not decrease. ■

Corollary 5.4 (MA for DFT contains no Zeno behaviour) The MA $\mathcal{M}_\mathcal{F}$ for a DFT $\mathcal{F}$ contains no *Zeno behaviour*.

Proof. This directly follows from Lemma 5.2 and Lemma 5.3. ■

Last, the state space of the MA can in principle be very large. The number of states is in the order of n^n where $n = |\mathrm{BE}_\mathcal{F}|$ is the number of BEs [ADH03].

Lemma 5.5 (Size of MA for DFT) Let $\mathcal{M}_\mathcal{F}$ be the MA for DFT $\mathcal{F}$. The state-space

5

size of $\mathcal{M}_\mathcal{F}$ is in the worst case factorial in the number of BEs and PDEPs of $\mathcal{F}$:

$$|S| \leq (|\mathsf{BE}_\mathcal{F}| + 2 \cdot |\mathsf{PDEP}_\mathcal{F}| + 1)!,$$

where $n!$ is the factorial of n.

Proof (sketch). The state space of $\mathcal{M}_\mathcal{F}$ is generated by considering all possible traces of BE failures and PDEP evaluations. The bound follows from the fact that there are $n!$ different permutations on n elements. The complete complexity estimation is given in Appendix C.2.1 on page 354. ■

5.2.2.2 Handling of SEQ

One way to reduce the state space a little bit is by better handling of SEQs. As an example, consider state s_7 from Example 5.3. This state is *invalid*, because SEQ R is violated here—indicated by F^R. Invalid states correspond to *invalid traces* and therefore should not be part of the resulting state space. We therefore remove invalid states and as a result, also do not need to explore such states further.

We adapt the rules for the transitions in Definition 5.4 accordingly.

Definition 5.5 (Adapted transition rules for SEQ) Let $\mathcal{F}$ be a DFT.
We extend the rule for *probabilistic transitions* as follows:

$$\frac{\ldots \quad \forall r \in \mathsf{SEQ}_\mathcal{F} : \mathsf{state}\big(\mathsf{PropDep}(\tau\, \mathsf{f}_{\sigma(v)_2})\big)_r = \mathsf{O}^r}{\mathsf{state}(\tau) \xhookrightarrow{v} \mu}.$$

We extend the rule for *Markovian transitions* similarly:

$$\frac{\ldots \quad \forall r \in \mathsf{SEQ}_\mathcal{F} : \mathsf{state}(\mathsf{PropDep}(\tau\, \mathbf{f}_v)) = \mathsf{O}^r}{\mathsf{state}(\tau) \overset{\lambda}{\dashrightarrow} \mathsf{state}(\mathsf{PropDep}(\tau\, \mathbf{f}_v))}.$$

We also need to extend the relation $\mathsf{NextDep}(\tau)$ to account for the fact that failure forwarding in dependencies might be enabled but is prohibited by SEQs:

$$\mathsf{NextDep}(\tau) = \exists v \in \mathsf{PDEP} : \ldots \wedge \forall r \in \mathsf{SEQ}_\mathcal{F} : \mathsf{state}\big(\mathsf{PropDep}(\tau\, \mathsf{f}_{\sigma(v)_2})\big)_r = \mathsf{O}^r.$$

Example 5.4 (Adapted MA for SEQ) We consider the MA from Example 5.3 depicted in Figure 5.4b. With the adapted transition rules for SEQ, state s_7 is never created. The premise of the transition rule is violated, because $\mathsf{state}(\tau_7)_R = \mathsf{F}^R$. Additionally, state s_3 has no outgoing Markovian transition with rate 6. BE D is not allowed to fail in s_3 as SEQ R would be violated.

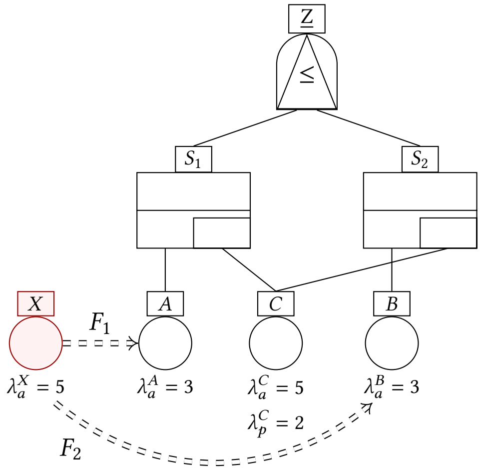
5.2.2.3 Non-determinism in FDEPs

Our state-space generation approach of *Storm* localises all non-determinism in the handling of dependencies. Non-determinism only occurs in the MA if failure forwarding is possible in multiple dependencies at the same time. In particular, *spare races* cannot occur, because they are always resolved beforehand by the choice which PDEP triggers first. As stated before in Section 4.4.3, only one SPARE is claiming at all times in *Storm* (cf. [Jun15, Theorem 4.30]).

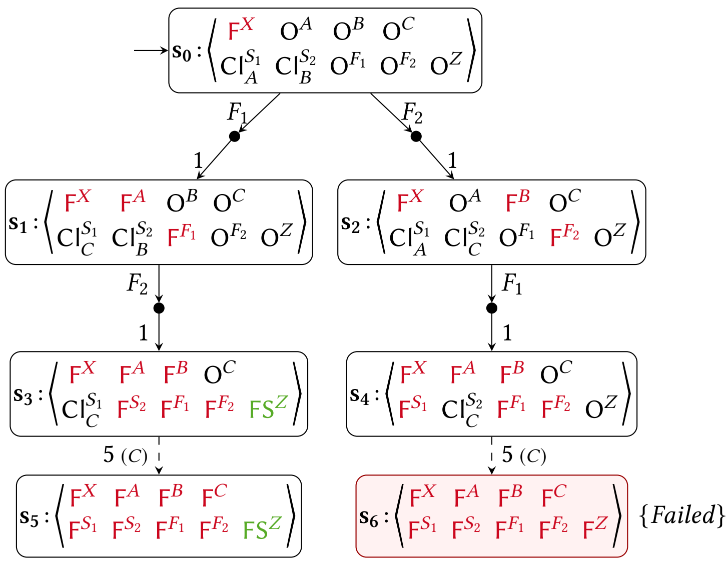
Example 5.5 (MA for DFT) We consider the example DFT $\mathcal{F}_2$ in Figure 5.5a which was already presented in Section 4.4.3. The DFT models a possible spare race due to the failure of X which triggers both A and B through dependencies F_1 and F_2, respectively. The corresponding MA is depicted in Figure 5.5b.
The initial state of the MA is state s_0 where X is already failed. From there on, both FDEPs F_1 and F_2 can be triggered. This is encoded in the MA by having two outgoing non-deterministic choices with actions F_1 and F_2, respectively. If for example FDEP F_1 triggers first, we go to state s_1 where SPARE S_1 has claimed the spare child C. From there on, FDEP F_2 is considered next and we end up in state s_3. In state s_3, SPARE S_2 has no available child for claiming and therefore fails. As a result, the top-level PAND Z becomes fail-safe. In contrast, taking FDEP F_2 first in state s_0 leads to states s_2, s_4 and s_6, subsequently. In state s_6, the TLE Z is failed. Note that the possible spare race between S_1 and S_2 over which SPARE claims the spare child C is resolved by moving this choice already to the FDEPs. Taking either F_1 or F_2 therefore determines which SPARE claims first.

5.3 Measures

We perform fault tree analysis using standard *model checking* techniques [BK08; CHVB18]. If the resulting model contains non-determinism, we obtain a *Markov automaton* which is analysed by probabilistic model checking [GHHK+14]. If the resulting model is deterministic, we built a *CTMC* instead. CTMCs allow to use more efficient model checking techniques tailored to this model type [BHHK03; BHHH+13]. Developing model-checking algorithms for continuous-time models and implementing them for maximal performance are an active field of research [BHHK15; BHH19; HHHK+19; BHKK+21]. We therefore do not go into the details of how the model checking is performed. We provide a model—MA or CTMC—and a property—the measure to compute—as input and the algorithm automatically computes the corresponding result.



(a) Example DFT $\mathcal{F}_2$ (restated from Figure 4.17 on page 165)



(b) MA $\mathcal{M}_{\mathcal{F}_2}$

Figure 5.5: Example for non-determinism in MA

5.3.1 Common measures

We use *continuous stochastic logic (CSL)* (cf. Section 2.3.1) to formalise the measures we are interested in for our DFT analysis.

Labels | We mainly use the label *Failed* to identify states where the TLE has failed, i.e. the complete DFT has failed. The corresponding *state formula* is $\Phi = \mathit{Failed}$. It is also possible to identify states where certain component failures have or have not occurred. This can for example be used to distinguish states where the primary component A of a SPARE has failed, but the spare component S is still operational. We identify such states with the state formula $\Phi = \mathit{Failed}_A \wedge \neg \mathit{Failed}_S$.

Measures | In DFT analysis, we are mostly interested in two measures: the *reliability* of a system and the *mean-time-to-failure (MTTF)*.

Definition 5.6 (Reliability) The *unreliability* of DFT element v within t time units is described by the CSL property: unreliability

$$\mathsf{Unreliability}_v^t = \mathbf{P}\left(\Diamond^{\leq t}\ \mathit{Failed}_v\right).$$

The *reliability* is given as the opposite of the unreliability: reliability

$$\mathsf{Reliability}_v^t = 1 - \mathsf{Unreliability}_v^t.$$

5

Definition 5.7 (Mean-time-to-failure (MTTF)) *Mean-time-to-failure (MTTF)* of DFT element v is described by the CSL property: mean-time-to-failure (MTTF)

$$\mathsf{MTTF}_v = \mathbf{ET}(\Diamond\ \mathit{Failed}_v) = \int_{t \geq 0} t \cdot \mathbf{P}\left(\Diamond^{[t,t]}\ \mathit{Failed}_v\right) \mathrm{d}t.$$

We omit the subscript v to denote the TLE *top*, e.g. $\mathsf{Unreliability}^t = \mathsf{Unreliability}_{top}^t$.

If the resulting model is an MA, we compute minimal and maximal values for the measures. In the presence of non-deterministic behaviour, we are often interested in the worst-case behaviour. For unreliability, we therefore compute the maximal unreliability and similarly, we compute the minimal MTTF.

Both measures can be extended to arbitrary state formulas instead of Failed_v. Moreover, we usually compute these measures starting from initial state s_0 where all components are operational. It is also possible to compute both measures from states where some components have already failed.

In general, using model checking offers great flexibility regarding fault tree analysis. We can check every property expressible in CSL without any modifications in the analysis algorithms. Further examples of more complex measures are given in Section 8.4.1.

5.3.2 Criticality of components

Common measure of interest for static fault trees are importance measures for components, i.e. 'how much impact has the failure of a component on the overall system reliability?' [DR01; TB17]. Probably the most used importance measure is the importance index by Birnbaum [Bir68].

Birnbaum importance index

Definition 5.8 (Birnbaum importance index) The *Birnbaum importance index* for DFT element v in DFT $\mathcal{F}$ at time t is given as

$$I_v^t = \frac{\partial \mathsf{Unreliability}_{top}^t}{\partial \mathsf{Unreliability}_v^t}.$$

The importance index describes the change in the overall system unreliability when changing the component unreliability. A high index—close to one—characterises highly critical components whereas a value near zero indicates that a failure of the corresponding component very rarely leads to a decrease in the system reliability.

For Markov models, an approximative computation of I_v^t is possible [OD03].

Definition 5.9 (Approximation of Birnbaum importance index) The *Birnbaum importance index* can be approximated by

$$\widetilde{I}_v^t = x \cdot \left(\frac{\mathbf{P}\left(\Diamond^{\leq t} \left(\mathit{Failed} \wedge \mathit{Failed}_v\right)\right)}{\mathbf{P}\left(\Diamond^{\leq t} \mathit{Failed}_v\right)} - \frac{\mathbf{P}\left(\Diamond^{\leq t} \left(\mathit{Failed} \wedge \neg \mathit{Failed}_v\right)\right)}{\mathbf{P}\left(\Diamond^{\leq t} \neg \mathit{Failed}_v\right)} \right),$$

where the factor

$$x = \frac{\mathsf{Unreliability}_v^t}{\mathsf{Unreliability}_{v_{iso}}^t}$$

determines the fraction of the unreliability of element v in the system and in isolation v_{iso}. The computation in isolation removes all restrictors, dependencies and parents of v and can therefore lead to a different result than the computation in the system.

5.4 Optimisations

We already have all necessary ingredients to perform fault tree analysis. However, the state space of the generated MAs can explode very fast, because we need to consider all possible ftraces. Even medium sized DFTs with 20 BEs[3] can already lead to an MA

[3] e.g. the multiprocessor computing system (MCS) [MPBV+06] in variant `mcs/cmt2_1_2_dp_x.dft`

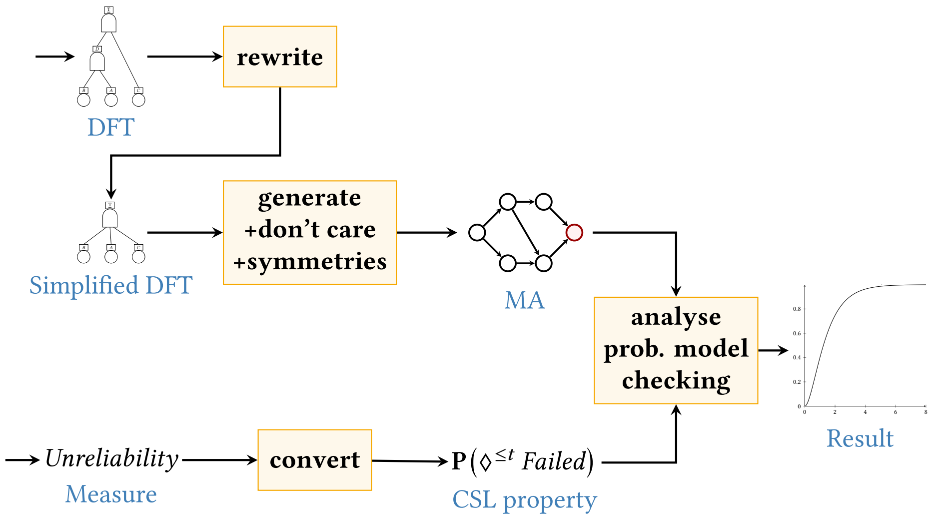


Figure 5.6: Overview of complete workflow for DFT analysis

with over 10 million states. The runtime of the analysis algorithms heavily depends on the number of states in the resulting MA. Thus, it is crucial to keep the resulting state space as small as possible. We therefore need to significantly improve the naïve approach in order to allow the analysis of larger DFTs.

In the following, we present several optimisations which are crucial for an efficient state-space generation. While the size of the MA can still be factorial in the number of BEs in the worst-case (cf. Lemma 5.5), our optimisations drastically reduce the resulting state space in practice. The analysis of DFTs is speeded up several orders of magnitude by employing the following optimisations. These improvements further allow to tackle DFTs which are otherwise not feasible. The actual performance improvements in practice will be evaluated in detail in Section 7.2.3.

Workflow with optimisations | We adapt the original workflow depicted in Figure 5.1 and incorporate several optimisations. The adapted workflow is depicted in Figure 5.6. During state-space generation we exploit several properties of the DFT. First, we ignore the failure of elements which become irrelevant during the exploration, i.e. we don't care about them any more. The details will be given in Section 5.4.1. We also exploit symmetries present in the DFT by applying symmetry reduction. The corresponding details will be given in Section 5.4.2. Depending on the given measure, we can split the DFT into several modules and analyse them independently, cf. Section 5.4.3. Additionally, we apply static analysis for dependencies to check whether the non-determinism stemming from these dependencies is *spurious*. We can resolve spurious non-determinism by fixing an order for the choices, cf. Section 5.4.4. Lastly, we simplify DFTs before the state-space generation by rewriting them in a preprocessing step, cf. Section 5.4.5.

Stop exploration | A simple optimisation is to stop exploration as soon as the TLE is failed. A state where the TLE is failed is called a *failed state*. Similarly, if the TLE is fail-safe in a state, we call it a *fail-safe state*. The common measures presented in Section 5.3.1 only consider *path formulas* until reaching a failed state. That means, the behaviour after reaching a failed state is irrelevant in our analysis. During state-space generation, we therefore do not further explore failed or fail-safe states. This can already reduce the state space quite significantly. Additionally, we also do not generate *invalid states*, cf. Section 5.2.2.2.

5.4.1 Don't Care

Our first optimisation is the introduction of another state for each DFT element v: *don't care (DC)*, denoted by $-^v$. An element is DC if its exact status—operational, failed or fail-safe—is irrelevant for our analysis purposes.

One example for the benefit of using don't cares is an OR-gate with multiple children. If one of the children fails, the OR-gate fails as well and its failure is propagated upwards. At this point, the failure of any of the other operational children does not change the state of the OR-gate any more—because it already failed. Thus, we 'don't care' about these other children any more, because their failure does not change anything. All these children are set to DC and we also do not investigate their failures, i.e. the corresponding transitions are not added to the MA. Note that elements can only be set to DC if their failure indeed does not change anything in the complete DFT. For example, elements with multiple parents can only be DC if all of their parents are either failed or fail-safe.

A second effect of using DC is the possible merge of several MA states. We again explain this advantage on the example of the OR-gate. If the OR-gate fails, we set all children—including the child which just failed—to DC. That way, all children have the same status. As a result, the failure of any of the children leads to the same state: the OR-gate is failed and all children are DC. Previously, we would have had one state for the failure of each child. The failure of the first child would lead to a state where only this child and the OR are failed and all other children are operational. In contrast, the failure of the last child would lead to a different state where the last child and the OR are failed. However, in both states all future behaviour would be the same, because the only relevant information is that the OR-gate has failed. By using don't cares we can therefore merge states with the same future behaviour into one unique state. This again can drastically reduce the state space.

We formalise the requirements for a don't care element in the following.

Definition 5.10 (Don't care element) A DFT element $v \neq top$ is *don't care* after trace τ in a DFT $\mathcal{F}$, i.e. $\mathsf{state}(\tau)_v = -^v$, if the following conditions hold: don't care

- For $v \in \mathcal{F} \setminus (\mathsf{BE}_\mathcal{F} \cup \mathsf{SPARE}_\mathcal{F} \cup \mathsf{PDEP}_\mathcal{F} \cup \mathsf{SEQ}_\mathcal{F})$:
$$\forall v' \in \rho(v) : \mathsf{state}(\tau)_{v'} \neq \mathsf{O}^v.$$
- For $v \in \mathsf{SPARE}_\mathcal{F}$:
$$\begin{aligned}&\forall v' \in \rho(v) : \mathsf{state}(\tau)_{v'} \neq \mathsf{O}^v\\ \wedge&\forall v' \in \sigma(v) : (\mathsf{state}(\tau)_{v'} \neq \mathsf{O}^v \vee \rho(v') \cap \mathsf{SPARE}_\mathcal{F} = \{v\})\,.\end{aligned}$$
- For $v \in \mathsf{BE}_\mathcal{F}$:
$$\forall v' \in \rho(v) : (\mathsf{state}(\tau)_{v'} \neq \mathsf{O}^v \vee (v' \in \mathsf{PDEP}_\mathcal{F} \wedge v = \sigma(v')_2))\,.$$
- For $v \in \mathsf{PDEP}_\mathcal{F}$:
$$\mathsf{state}(\tau)_{\sigma(v)_2} \neq \mathsf{O}^{\sigma(v)_2}.$$
- For $v \in \mathsf{SEQ}_\mathcal{F}$:
$$\forall v' \in \sigma(v) : \mathsf{state}(\tau)_{v'} \neq \mathsf{O}^v.$$

In general, a DFT element is don't care if all its parents are not operational any more, i.e. they are either failed, fail-safe or don't care. For SPAREs, additionally, issues with claiming of *shared* spares have to be prevented. A SPARE whose failure is irrelevant might still influence the overall behaviour by claiming a shared spare later on—which is then unavailable for other SPAREs. Thus, a SPARE can only be set to DC if all its operational children are not shared. Dependencies and restrictors also need special consideration for don't cares. If a dependent element of a dependency is not operational any more, triggering it will not lead to any change. Thus, the corresponding dependency can be set to DC in this case. Moreover, a dependent element—which can only be a BE—can become DC even though its corresponding dependency is still operational. If a dependent element is DC it does not matter whether it fails later, even if it is triggered by a PDEP. Lastly, a SEQ is don't care if all its children are not operational any more. In this case, its failure restriction was already successfully checked for all children before.

Don't cares are already supported by Definition 5.4 on page 187 without any modification. If an element v is don't care in a state s, the outgoing transition corresponding to the failure of v is not created. Note that this does not change the analysis results, but yields a smaller state space.

5

We show the correctness of using don't cares in the following. We restrict ourselves to CTMCs here and use the notion of *weak bisimulation* [BK08; BKHW05]. We do not consider Markov automata as several different notions of weak bisimulation exist, cf. [Tim13; EHZ10]. As the non-determinism occurs only in very restricted instances for DFTs, the results should easily translate to MAs as well.

Theorem 5.6 (Correctness of DC) Let $\mathcal{F}$ be a DFT (without dependencies), $C_{\mathcal{F}}$ be the corresponding CTMC and $C_{\mathcal{F}}^{DC}$ be the CTMC when using DC.
Then $C_{\mathcal{F}}$ and $C_{\mathcal{F}}^{DC}$ yield the same results for all measures w.r.t. the TLE, e.g. *unreliability* or *MTTF*. In fact, $C_{\mathcal{F}}$ and $C_{\mathcal{F}}^{DC}$ satisfy the same CSL properties (without the 'next' operator) when only using the label *Failed*.

Proof. We show that the CTMCs $C_{\mathcal{F}}$ and $C_{\mathcal{F}}^{DC}$ are *weakly bisimilar*. As weak bisimulation coincides with equivalence for CSL (without 'next' operator) [BKHW05], both CTMCs satisfy the same CSL properties w.r.t. to the label *Failed* of the TLE.
It remains to show that both CTMCs are indeed weakly bisimilar. Following [BKHW05, Definition 30], states $s, s' \in S$ are weakly bisimilar according to equivalence relation $R \subseteq S \times S$, denoted $(s, s') \in R$ if it holds:

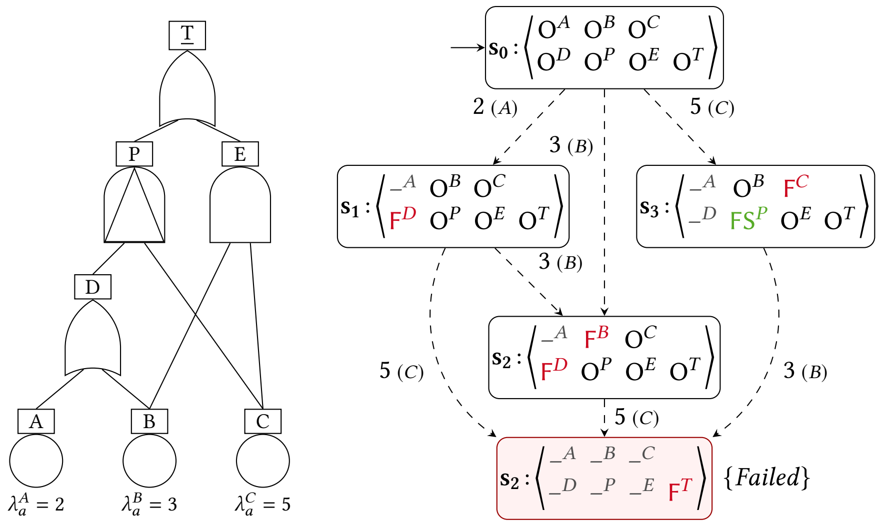
1. $L(s) = L(s')$ and
2. for each equivalence class D in S/R with $s \notin D$ it holds:

$$\sum_{\hat{s} \in D} \mathrm{rate}(s, \hat{s}) = \sum_{\hat{s} \in D} \mathrm{rate}(s', \hat{s}).$$

Let $\mathrm{state}^{DC}(\tau)$ be the state obtained from $\mathrm{state}(\tau)$ by applying don't cares. Let $R = \{(\mathrm{state}(\tau), \mathrm{state}(\tau') \mid \mathrm{state}^{DC}(\tau) = \mathrm{state}^{DC}(\tau'))\}$ be an equivalence relation on $C_{\mathcal{F}}$ which merges states which are equivalent according to DC.
We show that R is a weak bisimulation. Obviously, $\mathrm{state}^{DC}(\tau)_{top} = \mathrm{state}(\tau)_{top}$ for all traces τ, because the TLE is never DC. As *Failed* corresponding to *top* is the only labelling, it holds:

$$L(\mathrm{state}(\tau)) = L(\mathrm{state}^{DC}(\tau)) = L(\mathrm{state}^{DC}(\tau')) = L(\mathrm{state}(\tau)),$$

and the first condition holds.
Let $\mathbf{f}_A$ be an arbitrary SE failure which can happen in $\mathrm{state}(\tau)$, i.e. $\mathrm{state}(\tau)_A = \mathrm{O}^A$. If $\mathrm{state}^{DC}(\tau)_A = \mathrm{O}^A$, then also $\mathrm{state}(\tau') = \mathrm{O}^A$ and A can also fail in $\mathrm{state}(\tau')$. That means the rate for A is present in both states. If however, $\mathrm{state}^{DC}(\tau)_A = -^A$, then A might already be failed in $\mathrm{state}(\tau')$. In this case however, the failure of A in $\mathrm{state}(\tau)$ cannot make a difference—otherwise it would not be DC—and therefore $\mathrm{state}^{DC}(\tau\, \mathbf{f}_A) = \mathrm{state}^{DC}(\tau)$. Thus, $(\mathrm{state}(\tau, \mathbf{f}_A), \mathrm{state}(\tau)) \in R$. The successor state



(a) Example DFT $\mathcal{F}_3$ (b) MA $\mathcal{M}_{\mathcal{F}_3}$

Figure 5.7: Example for Don't Care in state-space generation

state(τ $\mathbf{f}_A$) is therefore in the same equivalence class as state(τ) and not considered in the second condition for the weak bisimulation. Thus, the second condition holds as well and R is a weak bisimulation. The CTMC $C^{DC}_{\mathcal{F}}$ is the weak bisimulation quotient of $C_{\mathcal{F}}$. ■

Example 5.6 (Don't Care propagation) We show the benefit of using don't cares for the example DFT $\mathcal{F}_3$ in Figure 5.7a.
First, we consider the failure of A, i.e. ftrace $\mathbf{f}_A$. The resulting state—without application of don't cares—is as follows:

$$\text{state}(\mathbf{f}_A) = \langle \mathsf{F}^A, \mathsf{O}^B, \mathsf{O}^C, \mathsf{F}^D, \mathsf{O}^P, \mathsf{O}^E, \mathsf{O}^T \rangle.$$

BE A can be marked don't care, because its only parent D is already failed. The resulting state is

$$\text{state}(\mathbf{f}_A)' = \langle -^A, \mathsf{O}^B, \mathsf{O}^C, \mathsf{F}^D, \mathsf{O}^P, \mathsf{O}^E, \mathsf{O}^T \rangle.$$

Next, we consider the failure of B in the initial state, i.e. ftrace $\mathbf{f}_B$. The resulting state—without don't cares—is as follows:

$$\text{state}(\mathbf{f}_B) = \langle \mathsf{O}^A, \mathsf{F}^B, \mathsf{O}^C, \mathsf{F}^D, \mathsf{O}^P, \mathsf{O}^E, \mathsf{O}^T \rangle.$$

Again, BE A can be marked don't care, because its only parent D is already failed. Note that A is don't care even though it has not failed itself. That also means we do not need to consider the failure of A for successor states. BE B cannot be don't care yet, because it has a second parent E which is still operational. The state with don't cares is:

$$\text{state}(\mathbf{f}_B)' = \langle -^A, \mathsf{F}^B, \mathsf{O}^C, \mathsf{F}^D, \mathsf{O}^P, \mathsf{O}^E, \mathsf{O}^T \rangle.$$

Lastly, we consider the failure of C in the initial state, i.e. ftrace $\mathbf{f}_C$, which yields:

$$\text{state}(\mathbf{f}_B) = \langle \mathsf{O}^A, \mathsf{O}^B, \mathsf{F}^C, \mathsf{O}^D, \mathsf{FS}^P, \mathsf{O}^E, \mathsf{O}^T \rangle.$$

Here, PAND P has become fail-safe. As a result, its first child D can be set to don't care, because the PAND can never fail. BE A can again be set to DC as well and the resulting state is:

$$\text{state}(\mathbf{f}_C)' = \langle -^A, \mathsf{O}^B, \mathsf{F}^C, -^D, \mathsf{FS}^P, \mathsf{O}^E, \mathsf{O}^T \rangle.$$

Using don't cares we obtain the complete state space $\mathcal{M}_{\mathcal{F}_3}$ as depicted in Figure 5.7b. The MA consists of only 5 states and 8 transitions. In contrast, the state space without don't cares is twice as large and has 11 states and 16 transitions.

Related work | Using some sort of don't care is a very natural idea for state-space reduction and was already used in other works.

Our definition of don't care is nearly identical to the trimming employed for *Boolean-logic driven Markov decision processes (BDMPs)* [BB03]. BDMPs are an extension of static fault trees and for example also support PAND-gates and a trigger which activates passive components—similar to activation in SPAREs. BDMPs employ a mechanism called trimming which ignores the failure of trimmed components—much like our don't cares. An element can be trimmed, if all its parents are failed and it has no outgoing trigger to an operational element [BB03, Section 4.2.3]. The definition of trimming is therefore very similar to our Definition 5.10. The main difference is that trimming does not handle fail-safe states—which could also be trimmed. Additionally, our approach also handles PDEP and SEQ, which are not present in BDMPs.

Yevkin presented an elimination of irrelevant states [Yev16] which conforms to our don't cares. The approach however only works for children of OR- and PAND-gates and is not generalised to all DFT elements.

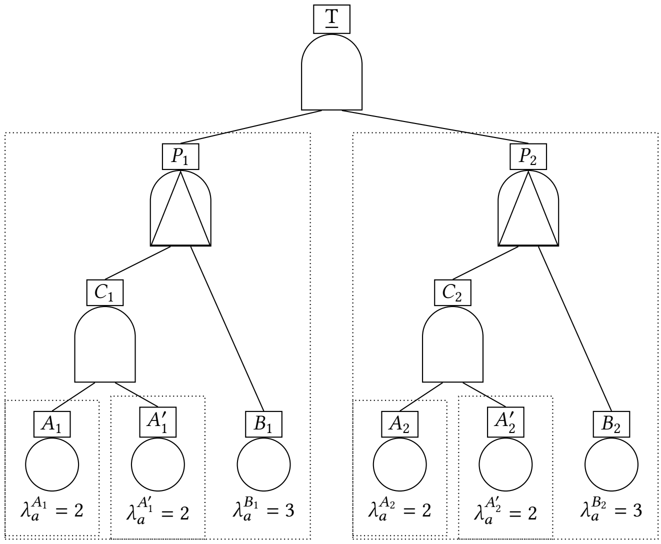


Figure 5.8: Example DFT $\mathcal{F}_4$

5.4.2 Symmetries

Safety-critical systems most often contain redundancies. In the corresponding DFTs this manifests as identical—or at least very similar—subtrees. Our next optimisation exploits these identical structures during state-space generation by performing *symmetry reduction* [CEJS98] within the DFT.

Example 5.7 (Symmetries in DFT) We use the DFT $\mathcal{F}_4$ in Figure 5.8 as an example. BEs A_1 and A_1' are symmetric. They are both under the same AND-gate and because the gate is static, their ordering does not matter. Furthermore, both BEs have the same failure rate. Both BEs are therefore *isomorphic*.
The complete subtrees for PANDs P_1 and P_2 are also symmetric. The first child C_1 of P_1 is symmetric to the first child C_2 of P_2 and similarly, B_1 is symmetric to B_2.

We give the formal definition for symmetric DFT elements.

Definition 5.11 (Symmetries in DFT) Let $\mathcal{F}$ be a DFT and $s_1, s_2 \in \mathcal{F}$. Let $S_1 = \text{subtree}(\mathcal{F}, s_1)$ and $S_2 = \text{subtree}(\mathcal{F}, s_2)$ be *subtrees* in $\mathcal{F}$ with top elements s_1 and s_2, respectively.
The subtrees S_1 and S_2 are *symmetric subtrees* if there is a bijective function $\text{sym}_{s_2}^{s_1} : V_{S_1} \to V_{S_2}$ from the elements of S_1 to S_2 such that the following holds:

·symmetric subtrees

1. Both top elements are symmetric and share the same static parent(s):

$$\mathsf{sym}_{s_2}^{s_1}(s_1) = s_2 \wedge \rho(s_1) = \rho(s_2) \subseteq \mathsf{StaticGates}_{\mathcal{F}}.$$

Additionally, for symmetric $v \in S_1$ and $v' \in S_2$, i.e. $\mathsf{sym}_{s_2}^{s_1}(v) = v'$, it must hold:

2. Symmetric elements have the same type:

$$\mathit{Type}(v) = \mathit{Type}(v').$$

3. Symmetric SEs have equal failure rates:

$$v, v' \in \mathsf{SE}_{exp} \implies \lambda_a^v = \lambda_a^{v'} \wedge \lambda_p^v = \lambda_p^{v'}.$$

4. Children of symmetric static gates are symmetric themselves or are the same:

$$v, v' \in \mathsf{StaticGates}_{\mathcal{F}} \implies \forall \hat{v} \in \sigma(v) : \exists \hat{v}' \in \sigma(v') : \left(\mathsf{sym}_{s_2}^{s_1}(\hat{v}) = \hat{v}' \vee \hat{v} = \hat{v}'\right).$$

5. Children of symmetric dynamic gates are symmetric themselves—or the same—and must respect the ordering:

$$v, v' \in \mathsf{DynamicGates}_{\mathcal{F}} \implies \bigwedge_{i=1}^{|\sigma(v)|} \left(\mathsf{sym}_{s_2}^{s_1}(\sigma(v)_i) = \sigma(v')_i \vee \sigma(v)_i = \sigma(v')_i\right).$$

6. Parents of symmetric elements are symmetric themselves or are the same:

$$\forall \hat{v} \in \rho(v) : \exists \hat{v}' \in \rho(v') : \left(\mathsf{sym}_{s_2}^{s_1}(\hat{v}) = \hat{v}' \vee \hat{v} = \hat{v}'\right).$$

Note that for VOT_k the equality of the threshold k for symmetric gates is ensured because k is part of the gate type $\mathit{Type}(v)$. Similarly, equality of probability p for PDEP_p is ensured through the *gate type* as well.

We denote the inverse function of the symmetric bijection with $\mathsf{sym}_{s_1}^{s_2} = \left(\mathsf{sym}_{s_2}^{s_1}\right)^{-1}$.

Example 5.8 (Symmetry relation) The DFT $\mathcal{F}_4$ of Example 5.7 contains three symmetries:

1. The subtrees of A_1 and A_1' are symmetric:

$$\mathsf{sym}_{A_1'}^{A_1}(A_1) = A_1'.$$

2. The subtrees of A_2 and A'_2 are symmetric:

$$\mathsf{sym}^{A_2}_{A'_2}(A_2) = A'_2.$$

3. The subtrees of P_1 and P_2 are symmetric:

$$\mathsf{sym}^{P_1}_{P_2}(P_1) = P_2, \qquad \mathsf{sym}^{P_1}_{P_2}(B_1) = B_2, \qquad \mathsf{sym}^{P_1}_{P_2}(C_1) = C_2,$$
$$\mathsf{sym}^{P_1}_{P_2}(A_1) = A_2, \qquad \mathsf{sym}^{P_1}_{P_2}(A'_1) = A'_2.$$

The symmetry $\mathsf{sym}^{A_2}_{A'_2}$ can be inferred from the symmetries $\mathsf{sym}^{A_1}_{A'_1}$ and $\mathsf{sym}^{P_1}_{P_2}$:

$$\mathsf{sym}^{A_1}_{A'_1}(A_1) = A'_1 \wedge \mathsf{sym}^{P_1}_{P_2}(A_1) = A_2 \wedge \mathsf{sym}^{P_1}_{P_2}(A'_1) = A'_2 \implies \mathsf{sym}(A_2) = A'_2.$$

During state-space generation we exploit the symmetries. In a state of the MA, we can swap the status of all DFT elements in a subtree with their symmetric counterpart.

Definition 5.12 (Symmetric state) Let $\mathcal{F}$ be a DFT and let S_1 and S_2 be symmetric subtrees of $\mathcal{F}$ with top elements s_1 and s_2, respectively. Let $\tau \in \mathsf{Traces}$ be a trace in $\mathcal{F}$.
The *symmetric state* of $\mathsf{state}(\tau)$ is obtained by swapping the status of symmetric elements: ·symmetric state

$$\mathsf{sym}^{s_1}_{s_2}(\mathsf{state}(\tau)_v) = \begin{cases} \mathsf{state}(\tau)_v & \text{if } v \notin S_1 \cup S_2, \\ \mathsf{state}(\tau)_{\mathsf{sym}^{s_1}_{s_2}(v)} & \text{if } v \in S_1, \\ \mathsf{state}(\tau)_{\mathsf{sym}^{s_2}_{s_1}(v)} & \text{if } v \in S_2. \end{cases}$$

For DFT element v, let v' be its symmetric element, i.e. $v' = \mathsf{sym}^{s_1}_{s_2}(v)$. The swapped status $s_{v'} = \mathsf{state}(\tau)_{\mathsf{sym}^{s_1}_{s_2}(v)}$ for status $s_v = \mathsf{state}(\tau)_v$ is defined as follows:

$$s_{v'} = \begin{cases} \mathsf{F}^{v'} & \text{if } s_v = \mathsf{F}^v, \\ \mathsf{O}^{v'} & \text{if } s_v = \mathsf{O}^v, \\ \mathsf{FS}^{v'} & \text{if } s_v = \mathsf{FS}^v, \\ -^{v'} & \text{if } s_v = -^v, \\ \mathsf{Cl}^{v'}_{\sigma(v')_i} & \text{if } s_v = \mathsf{Cl}^v_{\sigma(v)_i}. \end{cases}$$

The status of symmetric elements can be swapped. For example, if the original element v is operational and the symmetric element v' is failed, swapping the status makes v failed and v' operational. For SPAREs, if the i-th child is claimed, swapping lets the symmetric SPARE claims its i-th child.

5

Example 5.9 (Symmetric state) We continue with DFT $\mathcal{F}_4$ from Example 5.7. The initial state is given by

$$s_0 = \mathsf{state}(\varepsilon) = \langle \mathsf{O}^{A_1}, \mathsf{O}^{A'_1}, \mathsf{O}^{B_1}, \mathsf{O}^{C_2}, \mathsf{O}^{P_1}, \mathsf{O}^{A_2}, \mathsf{O}^{A'_2}, \mathsf{O}^{B_2}, \mathsf{O}^{C_2}, \mathsf{O}^{P_2}, \mathsf{O}^{T} \rangle.$$

Failure of BE A_1 yields the state

$$s_1 = \mathsf{state}\big(\mathbf{f}_{A_1}\big) = \langle \mathsf{F}^{A_1}, \mathsf{O}^{A'_1}, \mathsf{O}^{B_1}, \mathsf{O}^{C_2}, \mathsf{O}^{P_1}, \mathsf{O}^{A_2}, \mathsf{O}^{A'_2}, \mathsf{O}^{B_2}, \mathsf{O}^{C_2}, \mathsf{O}^{P_2}, \mathsf{O}^{T} \rangle.$$

Failure of BE A_2 yields a different state

$$s'_1 = \mathsf{state}\big(\mathbf{f}_{A_2}\big) = \langle \mathsf{O}^{A_1}, \mathsf{O}^{A'_1}, \mathsf{O}^{B_1}, \mathsf{O}^{C_2}, \mathsf{O}^{P_1}, \mathsf{F}^{A_2}, \mathsf{O}^{A'_2}, \mathsf{O}^{B_2}, \mathsf{O}^{C_2}, \mathsf{O}^{P_2}, \mathsf{O}^{T} \rangle.$$

However, because of the symmetries $\mathsf{sym}^{P_1}_{P_2}(A_2) = A_1$, we can swap the status of both subtrees and also obtain state s_1. Thus, $\mathsf{sym}^{P_1}_{P_2}(s'_1) = s_1$ and the ftraces $\mathbf{f}_{A_1}$ and $\mathbf{f}_{A_2}$ yield symmetric states.
Similarly, the following state is obtained after ftrace $\mathbf{f}_{B_1}\ \mathbf{f}_{A_2}\ \mathbf{f}_{A'_2}\ \mathbf{f}_{A'_1}$:

$$s_2 = \mathsf{state}\Big(\mathbf{f}_{B_1}\ \mathbf{f}_{A_2}\ \mathbf{f}_{A'_2}\ \mathbf{f}_{A'_1}\Big) = \langle \mathsf{O}^{A_1}, \mathsf{F}^{A'_1}, \mathsf{F}^{B_1}, \mathsf{O}^{C_2}, \mathsf{FS}^{P_1}, \mathsf{F}^{A_2}, \mathsf{F}^{A'_2}, \mathsf{O}^{B_2}, \mathsf{F}^{C_2}, \mathsf{O}^{P_2}, \mathsf{FS}^{T} \rangle.$$

Using the same symmetries as before, we obtain the symmetric state

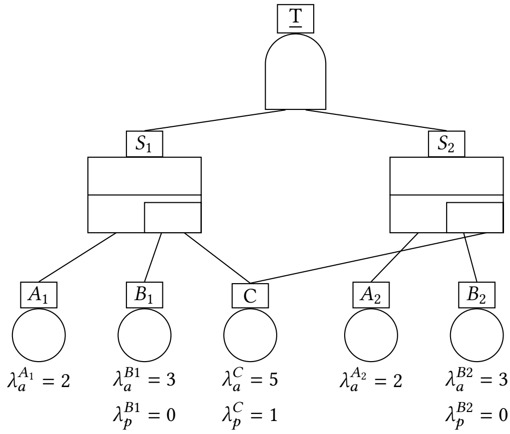
$$s'_2 = \mathsf{state}\Big(\mathbf{f}_{B_2}\ \mathbf{f}_{A_1}\ \mathbf{f}_{A'_1}\ \mathbf{f}_{A'_2}\Big) = \langle \mathsf{F}^{A_1}, \mathsf{F}^{A'_1}, \mathsf{O}^{B_1}, \mathsf{F}^{C_2}, \mathsf{O}^{P_1}, \mathsf{O}^{A_2}, \mathsf{F}^{A'_2}, \mathsf{F}^{B_2}, \mathsf{O}^{C_2}, \mathsf{FS}^{P_2}, \mathsf{FS}^{T} \rangle.$$

We can also exploit the symmetries between A_2 and A'_2 and obtain

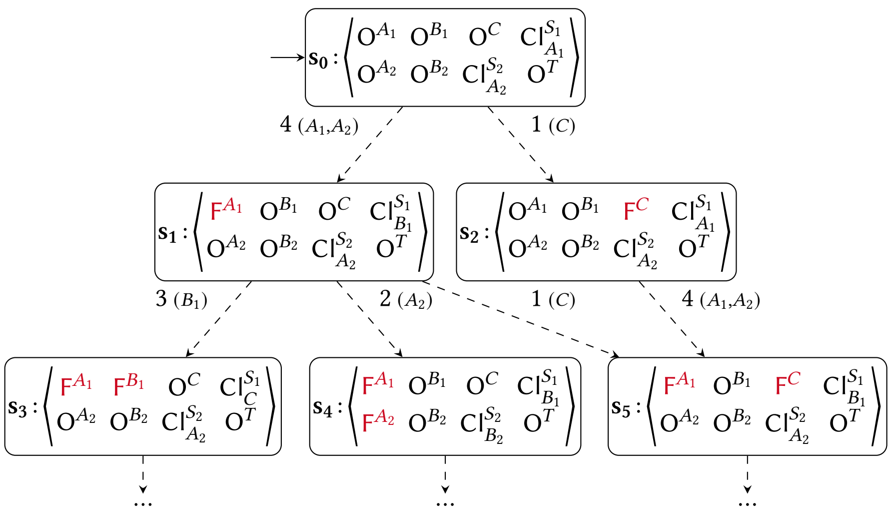
$$s''_2 = \mathsf{state}\Big(\mathbf{f}_{B_2}\ \mathbf{f}_{A_1}\ \mathbf{f}_{A'_1}\ \mathbf{f}_{A_2}\Big) = \langle \mathsf{F}^{A_1}, \mathsf{F}^{A'_1}, \mathsf{O}^{B_1}, \mathsf{F}^{C_2}, \mathsf{O}^{P_1}, \mathsf{F}^{A_2}, \mathsf{O}^{A'_2}, \mathsf{F}^{B_2}, \mathsf{O}^{C_2}, \mathsf{FS}^{P_2}, \mathsf{FS}^{T} \rangle.$$

During state-space generation, we can merge symmetric states and therefore reduce the resulting state space. Note that don't cares and symmetries complement each other and using both optimisations leads to even further state-space reductions. The DFT of Example 5.9 for example yields 11 states when using both symmetries and don't cares compared to 26 states when only using don't cares.

One crucial advantage of our definition of symmetries is that symmetric subtrees can 'overlap'. Symmetric elements can have the same children or parents and do not need to be completely *independent*. One example are *shared* spares which are children of multiple SPAREs. Our approach exploits the symmetric structure often present in redundancies modelled by SPAREs even though the SPAREs are still connected by shared spares. As a result, we allow for more and larger symmetries in DFTs which in turn yield smaller state spaces.



(a) Example DFT $\mathcal{F}_5$



(b) MA $\mathcal{M}_{\mathcal{F}_5}$

Figure 5.9: Example for symmetries with shared spares

5

Example 5.10 (Symmetries in shared spares) We depict DFT $\mathcal{F}_5$ with shared spares in Figure 5.9a. Both SPAREs S_1 and S_2 are symmetric:

$$\mathsf{sym}^{S_1}_{S_2}(S_1) = S_2 \qquad \mathsf{sym}^{S_1}_{S_2}(A_1) = A_2 \qquad \mathsf{sym}^{S_1}_{S_2}(B_1) = B_2$$

Shared spare C is the third child of both S_1 and S_2. Therefore the conditions of Definition 5.11 hold, because both symmetric elements have the same child. The (partial) state space of the resulting MA $\mathcal{M}_{\mathcal{F}_5}$ is given in Figure 5.9b. From the initial state s_0, the failure of A_1 leads to state s_1 with failure rate 2. Similarly, the failure of the symmetric BE A_2 first leads to state

$$s_1' = \mathsf{state}(\mathbf{f}_{A_2}) = \langle \mathsf{O}^{A_1}, \mathsf{O}^{B_1}, \mathsf{O}^{C}, \mathsf{Cl}^{S_1}_{A_1}, \mathsf{F}^{A_2}, \mathsf{O}^{B_2}, \mathsf{Cl}^{S_2}_{B_2}, \mathsf{O}^{T} \rangle.$$

Applying the symmetries swaps status O^{A_1} and F^{A_2}. Additionally, the status of the SPARE gates is also swapped: $\mathsf{Cl}^{S_1}_{A_1}$ and $\mathsf{Cl}^{S_2}_{B_2}$ are switched to $\mathsf{Cl}^{S_1}_{B_1}$ and $\mathsf{Cl}^{S_2}_{A_2}$. Note that swapping changes the claimed child for both SPAREs. The resulting state is s_1, i.e. $\mathsf{sym}^{S_1}_{S_2}(s_1') = s_1$ which is already present. The Markovian transition from s_0 to s_1 is therefore extended with the failure rate 2 of BE A_2 and in total has a rate of 4. The complete MA has 14 states which is significantly less than the 23 states without symmetry reduction.

Related work | Symmetry reduction is a common technique in model checking to alleviate the state-space explosion problem [CEJS98]. Symmetries have also been (partially) exploited for DFTs already. *DFTCalc* for example recognizes identical independent subtrees in the DFT and builds the corresponding I/O-IMC only once and reuses this result [BCS10].

The approach in [BC04] is closest to our symmetry reduction. They extend DFTs with a replicator event which is used to compactly represent identical subtrees and can also handle shared spares. This compact representation is used in their underlying model, stochastic well-formed nets (SWNs). The symmetry inherent in the replicator event is exploited during analysis. '[T]he analysis is based on the translation into a SWN that exploits the high level of symmetry [...] by directly generating the CTMC in a lumped form.' [BC04] The authors provide an example analysis of a highly redundant system (based on [ZZDS01]) which yields a state-space size of 35 states [BC04]. Our approach yields a state-space size of 10 states for the same DFT and therefore further improves on existing state-space reduction techniques.

In model checking, state-space reduction techniques can be applied after the model is built to minimise models. One such technique is *bisimulation minimisation* [BK08]. Applying bisimulation on the MA also exploits—among others—the symmetric structures present in the model. However, this minimisation can only be performed after the model has been generated. It therefore only helps in reducing the time required for computing the measures but not the time for initially building the model. In contrast, our symmetry reduction is applied during state-space generation and therefore avoids building a larger (intermediate) state space.

5.4.3 Modularisation

Modularisation is a commonly used technique in fault tree analysis [GD97; AS98; BC04]. The idea is to analyse independent parts of the DFT separately and combine these results into the final result. These independent parts of a DFT are often called *modules*. To avoid confusion with the *spare module*, we refer to them as *independent subtrees* in the following.

Definition 5.13 (Independent subtree) Let $\mathcal{F}$ be a DFT and $v \in \mathcal{F}$ be a DFT element. A child $v' \in \sigma(v)$ of v is the top element of an *independent subtree* of v if $\mathsf{subtreeExtended}(\mathcal{F}, v')$ is a *real subtree*, cf. Definition 2.14 on page 24. ·independent subtree

If the subtree for element v' is a real subtree, v' is the root of this subtree. That means, there can be no shared element with subtrees of the other children of parent v. We call an independent subtree a *dynamic subtree* if it contains at least one dynamic gate. Otherwise, it is a *static subtree*.

5

Example 5.11 (Independent subtrees) We consider DFT $\mathcal{F}_6$ in Figure 5.10. TLE T has two independent subtrees: U and X. This can also be easily seen in the visualisation, because both subtrees (indicated by the boxes) do not overlap. OR U does not have independent subtrees, because the *extended subtrees* of K and L are the same due to their common child M. In other words, both subtrees 'overlap'.

Independent subtrees can be computed in linear time [DR96]. Assume we need to compute a *probability property* such as the *unreliability* $\mathbf{P}(\Diamond^{\leq t} \mathit{Failed}_{top})$ for a complete DFT with static TLE *top*. Instead of analysing the complete DFT, we calculate the result for each independent subtree with top element $\sigma(top)_i$ separately, i.e. $\mathbf{P}(\Diamond^{\leq t} \mathit{Failed}_{\sigma(top)_i})$. We combine the sub-results and calculate the overall result:

- If *top* is an AND, we compute:

$$\mathbf{P}(\Diamond^{\leq t} \mathit{Failed}) = \prod_{i=1}^{|\sigma(top)|} \mathbf{P}(\Diamond^{\leq t} \mathit{Failed}_{\sigma(top)_i}).$$

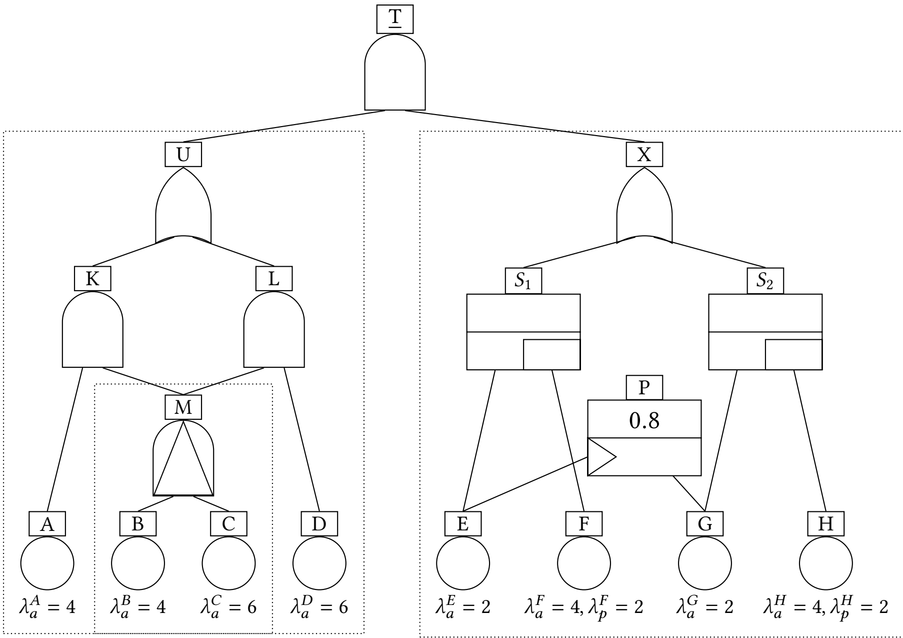


Figure 5.10: Example DFT $\mathcal{F}_6$

- If *top* is an OR, we compute:

$$\mathbf{P}\left(\Diamond^{\leq t} \textit{Failed}\right) = 1 - \prod_{i=1}^{|\sigma(top)|} \left(1 - \mathbf{P}\left(\Diamond^{\leq t} \textit{Failed}_{\sigma(top)_i}\right)\right).$$

As we only need to generate the state spaces for the smaller subtrees—and not the complete MA—the analysis time significantly decreases. This approach can also be applied recursively, if independent (static) subtrees again contain further subtrees.

Example 5.12 (Indepedent subtrees) We continue with Example 5.11 and compute the reliability of the DFT within 1 time unit. We compute the unreliability for both subtrees:

$$\mathsf{Unreliability}_U^1 = 0.398, \qquad \mathsf{Unreliability}_X^1 = 0.972.$$

The overall probability is the product of both sub-results:

$$\mathsf{Unreliability}^1 = 0.398 \cdot 0.972 = 0.387.$$

For both subtrees we generate MAs with 11 states and 7 states, respectively. This is a significantly smaller state space than the 71 states without modularisation.

Note that modularisation is not easily applicable to an *expected-time property*, because expectations would need to be combined in this case. One possible approach is to use the algorithm of [BH02] which computes the mean value of the maximum of phase-type distributed random variables, i.e. the MTTF for an AND-gate. Instead, we use an adapted version of modularisation for MTTF.

Adapted modularisation for expected-time properties | We assume in the following, that the DFT does not exhibit non-determinism and the resulting model is a CTMC. The idea of *adapted modularisation* is that we generate the state space for both subtrees in parallel and combine them in the end. We first generate the CTMCs for each independent subtree. Then, our procedure uses a dedicated algorithm to create the parallel construction of the CTMCs, which is given by the Cartesian product. If the TLE *top* is an AND, we label a state in the Cartesian product with *Failed* if this state has the labels $\mathit{Failed}_{\sigma(top)_i}$ for each subtree. For an OR, we label a state if the label $\mathit{Failed}_{\sigma(top)_i}$ of at least one subtree is present. Note that the construction of the Cartesian product does not need to create outgoing transitions for *Failed* states as we stop exploration there.

This algorithm can be improved by first reducing the single CTMCs by means of bisimulation minimisation [BK08; BHHK03] and applying the composition afterwards—akin to [HK00]. The algorithm can be executed recursively as well which further improves the state-space generation for subtrees. For isomorphic subtrees, the CTMCs are amenable to a form of symmetry reduction called counting abstraction [GS92]: it saves how many subtrees are in the same CTMC state.

In most cases, adapted modularisation provides only small improvements on the runtime of DFT analysis, because we still need to generate a quite large state space in the end. In contrast, if modularisation is applicable—i.e. we have a probability property and independent subtrees—it can decrease the analysis time by orders of magnitude. A thorough evaluation is provided in Section 7.2.3.

Related work | Most other works [GD97; BC04], simply replace each dynamic independent subtree with a BE whose failure probability corresponds to the failure probability of the subtree. The resulting tree is then a static fault tree which can be analysed with standard techniques for SFT analysis [RS15], for example with binary

5

decision diagrams (BDDs) [Rau93]. Recently, [KASP+20] presented a recursive modularisation where independent static subtrees are evaluated via an algebraic approach. Dynamic subtrees following an exponential distribution are converted into Petri nets in a translation which is very similar to our work in Chapter 3. Subtrees with other types of distributions are solved via Monte Carlo simulation.

The *adapted modularisation* is similar to the underlying approach of *DFTCalc* [BCS10]. While *DFTCalc* applies parallel composition for each two DFT elements, we restrict ourselves to composition of complete subtrees. That way, we exploit the independence of the subtrees and profit from fast (direct) state-space generation for each subtree.

5.4.4 Static analysis for dependencies

In probabilistic model checking, the analysis of *Markov automata* is more complex and therefore more time-consuming than the analysis of *CTMCs* [HHHK+19; BHKK+21]. This is in particular true for time-bounded properties such as *reliability*. The culprit for the computational effort is the presence of non-determinism in MAs. It is therefore desirable to obtain a CTMC instead of an MA after the translation from a DFT. In our setting, the source of non-determinism is located in the handling of PDEPs. Whenever multiple PDEPs can fire simultaneously, we obtain a non-deterministic choice, cf. Section 5.2.2.3.

In some instances, we are able to remove PDEPs and replace them by OR-gates, cf. Section 4.1.2. In all other instances, we can apply a pre-processing step to check whether non-deterministic behaviour is actually possible. For example, in case of static gates, the order in which children fail is irrelevant. That means, if multiple PDEPs are triggered simultaneously, the order in which they forward the failure to children of static gates is irrelevant. On the other hand, if a dependent event of an PDEP is child of a priority gate or a SPARE, the order of failure forwarding is crucial, because it can decide between failed and fail-safe, or lead to different resolutions of spare races.

We perform a static analysis on the DFT to determine which dependencies are in *conflict*, i.e. the order in which they fire eventually leads to different states in the DFT. PDEPs which are not in conflict can fire in a fixed order during state-space generation. As a result, we get only one choice in a state—the next PDEP according to the fixed order—instead of all possible choices.

Example 5.13 (Conflicts for dependencies) We give an exemplary depiction of FDEP conflicts for DFT $\mathcal{F}_7$ in Figure 5.11. The FDEPs F_3 and F_4 are in conflict, because they both trigger a child of a SPARE. Thus, they both trigger a spare race between S_1 and S_2 and we need to examine both possible orders for F_3 and F_4. In contrast, FDEPs F_1 and F_2 are not in conflict as they both trigger BEs which are in a *static subtree*. Thus, we fix an order and always evaluate F_1 before F_2.

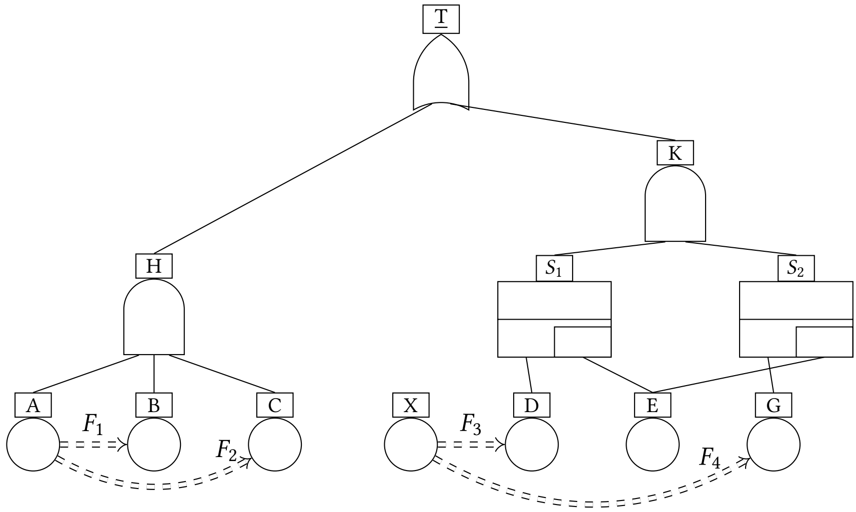
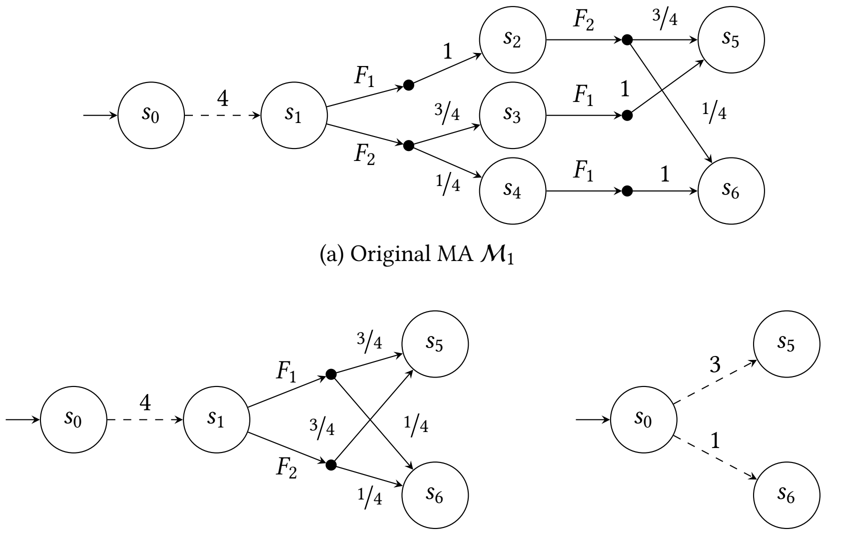


Figure 5.11: Example DFT $\mathcal{F}_7$

Eliminating spurious non-determinism | The static analysis is conservative and might mark PDEPs as being in conflict even though they are actually not. Thus, the resulting state space might still contain *spurious* non-determinism. However, we remove such non-determinism by applying a state elimination approach inspired by [Daw04]. We explain the idea by means of an example.

Example 5.14 (State elimination for non-determinism) We depict the state elimination for non-determinism in Figure 5.12. We start with the MA $\mathcal{M}_1$ in Figure 5.12a. All three states s_2, s_3 and s_4 have only one outgoing choice. We can therefore eliminate each state by directly routing each ingoing transition to the successor states of this state. For example, state s_2 has one ingoing transition from s_1 with action F_1 and probability 1. The outgoing transition with action F_2 goes to state s_5 with probability $3/4$ and to state s_6 with probability $1/4$. We eliminate state s_2 by adding a direct transition from s_1 with action F_1 to state s_5 with probability $1 \cdot 1/4$ and to state s_6 with probability $1 \cdot 3/4$. Similarly, states s_3 and s_4 are eliminated as well. The resulting MA $\mathcal{M}_2$ is depicted in Figure 5.12b.
In $\mathcal{M}_2$, we have two possible actions F_1 and F_2. However, both actions have identical outcomes. Thus, we merge them into one action. Next, we eliminate the remaining state s_1—which now has one outgoing transition. The probabilities of the outgoing probabilistic transition are multiplied with the rate of the ingoing



(a) Original MA $\mathcal{M}_1$

(b) MA $\mathcal{M}_2$ after eliminating s_2, s_3 and s_4

(c) CTMC $\mathcal{C}_3$ after eliminating s_1

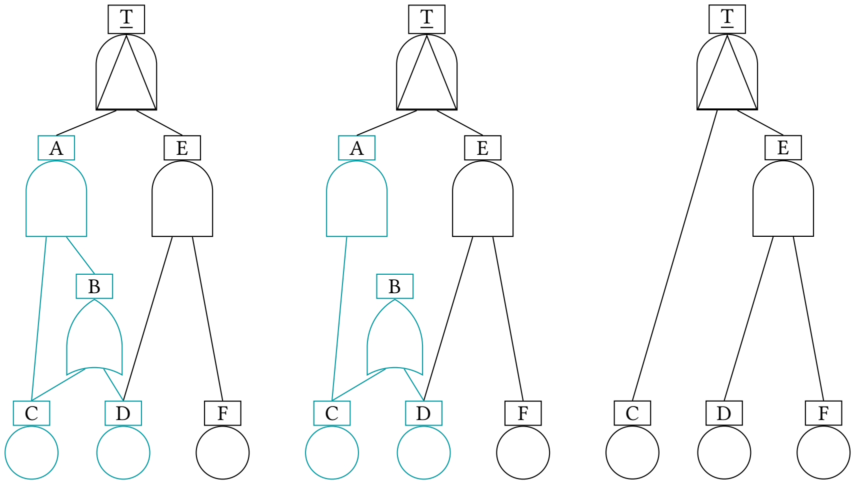
Figure 5.12: State elimination for non-determinism

Markovian transition. From state s_0 we reach state s_5 with rate $4 \cdot 3/4$ and we reach state s_5 with rate $4 \cdot 1/4$. The resulting CTMC $\mathcal{C}_3$ is depicted in Figure 5.12c. Note that we obtain a CTMC, because all non-determinism was eliminated.

Note that eliminating states needs careful handling of the labels. Depending on the property, labels of the eliminated states might either be removed as well or be added to the successor or predecessor states.

Using state elimination, we remove certain non-deterministic actions and simplify the resulting state space. If all non-determinism is *spurious*, we obtain a CTMC after state elimination. As a result, we can apply more efficient model-checking algorithms.

Related work | Our static analysis of dependency conflicts is related to confluence reduction for MAs [TPS13]. Confluent transitions connect bisimilar states and are always prioritised over the other outgoing transitions. Confluence reduction therefore fixes certain actions and thus, reduces the resulting state space. Our approach as well as confluence reduction are related to partial-order reduction for non-probabilistic systems [BK08].



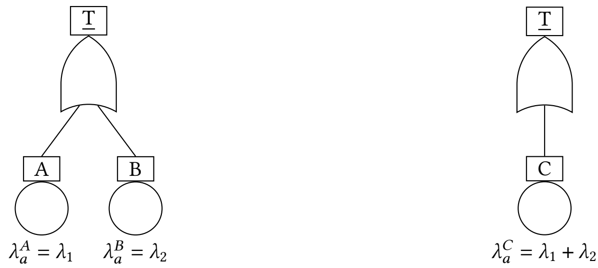
(a) Original DFT $\mathcal{F}_1^R$ (b) DFT $\mathcal{F}_2^R$ after rewrite step (c) Final DFT $\mathcal{F}_3^R$

Figure 5.13: Example application of rewrite rules

5.4.5 Rewriting

The last optimisation is applied even before the state-space generation takes place. A DFT can be simplified before the analysis by transforming the underlying *graph* of the DFT according to *rewrite rules* [Jun15; JGKR+17]. We explain the approach by means of an example.

Example 5.15 (Application of rewrite rules) The simplification of a DFT by rewriting is depicted in Figure 5.13. We start with the original DFT $\mathcal{F}_1^R$ in Figure 5.13a. The highlighted part of the DFT matches the structure of one of the rewrite rules. The corresponding rule handles subsumption of OR-gates by AND-gates [JGKR+17, Rewrite rule 8]. In the given case, the first input C is the only relevant input to AND A, because the failure of C also renders the second input—OR-gate B—failed. We apply the rewrite rule by replacing the matched part with the result of the rewrite rule. The resulting DFT $\mathcal{F}_2^R$ is depicted in Figure 5.13b. In $\mathcal{F}_2^R$, the second input of the AND-gate is removed as it is irrelevant.
Following this approach, we apply other rewrite rules which eliminate static gates with single inputs—AND-gate A—and elements which have no connection to the TLE—OR-gate B. The final DFT $\mathcal{F}_3^R$ is given in Figure 5.13c. Obviously, the resulting DFT is structurally simpler than the original one and also contains two less elements.



(a) Original DFT $\mathcal{F}_4^R$ (b) Final DFT $\mathcal{F}_5^R$

Figure 5.14: Merging of BEs

Junges developed 29 rewrite rules for simplification of DFTs [Jun15]. The rewrite rules handle a broad spectrum of DFT structures:

- structural identities, e.g. commutativity of static gates, removal of gates with a single successor or no predecessor, or left-flattening of gates,
- standard axioms from Boolean algebra, e.g. subsumption of OR by AND,
- handling of constant elements $\mathsf{CONST}_\top$ and $\mathsf{CONST}_\bot$,
- special cases for VOT_k-gate, e.g. translating VOT_1 and VOT_n into OR and AND, respectively,
- handling of PANDs, e.g. flattening or simplifying conflicting PANDs,
- handling of FDEPs, e.g. translation into OR or removal of superfluous FDEPs,
- non-structural rules, e.g. removing BEs with no path to the TLE or merging BEs.

We shortly present the merging of BEs as this is one of the most used rewrite rules and yields significantly simpler DFTs.

Merging BEs | In our analysis, we are mostly interested in the *reliability* or *MTTF* of the TLE. Similar to *don't care*, it is for example irrelevant which BE under a OR-gate has failed. We can therefore simplify a DFT by merging multiple BEs into a single BE. An example is given in Figure 5.14. The two BEs in Figure 5.14a have exponential failure distributions with rates λ_1 and λ_2, respectively. The failure distribution of an OR-gate is the minimum over its inputs and therefore it is exponentially distributed as well. Thus, we can replace both BEs be a single BE C in Figure 5.14b. This new BE C has a failure rate which is the sum of the original failure rates, i.e. $\lambda_a^C = \lambda_1 + \lambda_2$.

Restrictions | Note that some of the rewrite rules can only be applied in certain contexts, for example translating an FDEP into an OR, cf. Proposition 4.1 on page 140. The definitions of the rewrite rules take these context restrictions into account.

Note also that the rewrite framework is not confluent. That means, the order in which rewrite rules are applied heavily influences the resulting DFT and in particular the size of the DFT. A good heuristic is necessary to generate DFTs which are as small as possible [JGKR+17].

Experimental evaluation | Simplifying the DFT beforehand can significantly improve the scalability of the analysis approach. The experimental evaluation presented in [JGKR+17] indicates that rewriting heavily improves the performance of the DFT analysis. For example, originally 68% of the 183 DFT examples in [JGKR+17] were solved within 2 hours. After simplification through rewriting, 95% of the DFTs could be analysed. In addition, the total time needed for analysis was halved from 41 hours to 18 hours. In conclusion, simplifying DFTs by rewriting allows to analyse DFT which were out of reach before. Moreover, it leads to speed-ups and memory savings of up to two orders of magnitude [JGKR+17].

Correctness | The correctness of the rewrite rules was first proven by hand in [Jun15]. We later formally verified the rewrite rules using higher-order-logic (HOL) theorem proving [7]. The proofs are based on existing formalisations of DFTs [EAHT19] in the HOL4 theorem prover [SN08]. Using HOL4 allowed us to construct rigorous, mechanically checkable proofs for the correctness of the rewrite rules. Note that 7 rules handling FDEPs and SPAREs were not verified as these gates are not yet formalised in HOL4. Additionally, the correctness of the rewrite rules depends on a *bottom-up* evaluation order. This is not a problem though as *Storm* uses this evaluation order.

Summary | Rewriting is a helpful pre-processing step for DFT analysis. It allows the users the greatest flexibility when creating DFTs without impacting the analysis performance. In particular, no manual adaptation for better performance is necessary. The rewriting step fully automatically translates the given DFT into a structure which is best suited for analysis.

5

Summary

- ➜ DFT analysis is performed by translating a DFT into a corresponding Markov automaton (MA).
- ➜ States in the MA are given by the status—operational, failed, fail-safe—of each DFT element and the claiming status of the SPAREs.
- ➜ Successor states in the MA are created by letting a BE fail—either on its own or through dependencies—and propagating this failure upwards in the DFT.
- ➜ The decision which dependency forwards a failure first is encoded as a non-deterministic choice in the MA.
- ➜ Standard techniques from probabilistic model checking are employed to analyse the resulting MA. Common measures such as reliability and mean-time-to-failure (MTTF) as well as more intricate and complex measures are directly supported by the existing model checking algorithms.
- ➜ Several optimisation techniques are applied to reduce the state-space size and allow for faster analysis.
- ➜ Elements are marked as 'don't care' in a state if their failure does not influence the overall system anymore. Using don't cares avoids exploring irrelevant failures and can drastically reduce the state space.
- ➜ Symmetry reduction exploits redundancies in DFTs. Symmetric subtrees allow to consider the failure of a symmetric element instead of the original element and thus, lump symmetric states during state-space generation.
- ➜ State elimination is used to remove spurious non-determinism in the resulting state space.
- ➜ DFT rewriting can greatly simplify the structure of the DFT before analysis.

6 Partial State-Space Generation

Assumptions

In this chapter, we make the same assumptions as in Chapter 5. The DFT is *conventional*. We use the inclusive variants $\mathrm{PAND}_{\leq}$, $\mathrm{POR}_{\leq}$ and $\mathrm{SEQ}_{\leq}$, and use binary PDEPs.

The state-space generation presented in Chapter 5 already provides a good approach for DFT analysis. In particular, due to the optimisations detailed in Section 5.4, the presented approach can analyse most example DFTs from the literature, cf. Section 7.2. The analysis however becomes challenging for increasing numbers of BEs and if most of the optimisations can not be applied, e.g. no *symmetries* and no independent subtrees for *modularisation* are present. In these cases, the number of states which need to be generated can be very large. This phenomenon is called *state-space explosion* and is stated as 'the' problem of Markov-based analysis [BD05a; ZMFW09; XTD12; EHAT18; RRBS19]. In fact, most other approaches motivate the need for new analysis methods, because the 'state space becomes too large for calculation with Markov models when the number of gate inputs increases' [RGRK+09].

We present an approach based on partial state-space generation which aims to mitigate the state-space explosion. By building only the most relevant fragment of the state space, we obtain a good approximation of the actual result. The iterative nature of the refinement algorithms allows to approximate the exact value up to the desired precision. Moreover, the user is in control of the trade-off between computation time and precision of the result. The main advantage of this approach is that it computes sound bounds, i.e. the exact result always lies within the computed over- and under-approximation. This property is crucial in order to guarantee that the complete behaviour of the system is correctly assessed. The sound bounds are in contrast to other approximation methods that can give no formal guarantees about the closeness of the result. For example, simulation [RGRK+09; RRBS19; BBMD+20] only yields confidence intervals and the discretisation in Bayesian networks [BD05b; MPBV+06] also introduces inaccuracies. We argue that especially for safety-critical systems, hard formal guarantees on the correctness of the computed results are essential.

This chapter is based on the publication 'Fast Dynamic Fault Tree Analysis by Model Checking Techniques' [6].

6

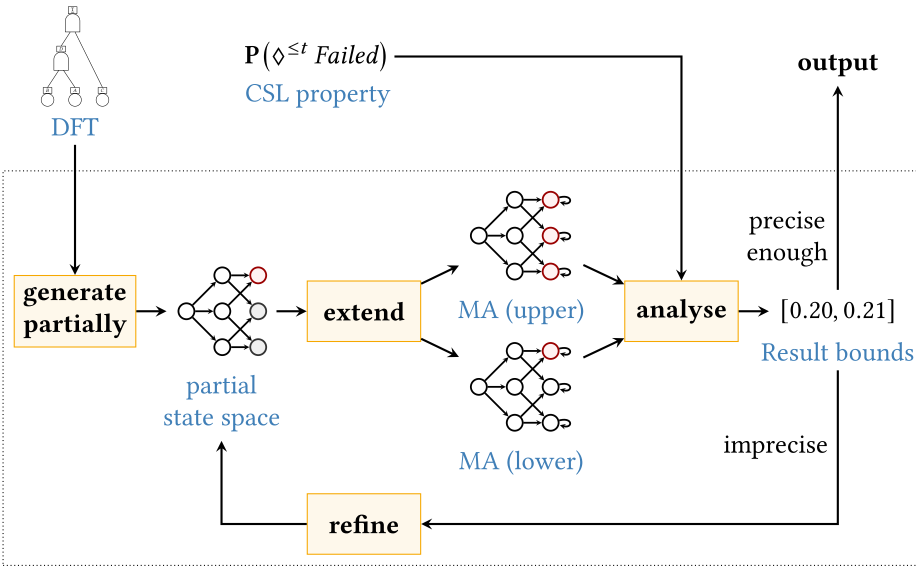


Figure 6.1: Overview of workflow for approximation in DFT analysis

6.1 Approximation in DFT analysis

The general idea of the approximation approach is depicted in Figure 6.1. We compute for example the *unreliability* of the system. Starting from a DFT, we generate a partial state space by stopping exploration at some point, i.e. no further BE failures are considered. The partial state space is extended to an MA for the upper bound–on the unreliability—by assuming that all omitted BEs have failed. Similarly, the MA for the lower bound is obtained by assuming all unexplored BEs will never fail. We analyse both MAs for the given CSL property and obtain two values: an upper bound u of the unreliability for one MA and a lower bound l for the other MA. The actual unreliability value will always lie within the interval $[l, u]$. If the interval is already precise enough, i.e. $u - l < \varepsilon$ for some given precision ε, we stop the approximation and output the interval. If the interval is still too imprecise, we refine the state space by exploring more states and start a new iteration.

The algorithm for the approximation is given in Algorithm 1. Depending on the property to check—either *unreliability* or *MTTF*—we initialise the bounds differently (Line 6). Then we perform the iterative approximation as long as the difference $u - l$ between the upper and lower bound is greater than the given error bound ε (Line 14). In each iteration, we extend the existing state space S by further exploring states from the queue Q (Line 9). Using the refined state space, we build the MAs for the upper

Algorithm 1 Approximation algorithm

```
 1: function AnalyseApproximation(DFT 𝓕, Property prop, Error bound ε)
 2:     S ← ∅                                          ▷ Initialise state space
 3:     s0 ← InitState(𝓕)                              ▷ Set initial state
 4:     Q ← {s0}                                       ▷ Initialise exploration queue
 5:     l ← 0
 6:     if prop is MTTF then  u ← ∞                    ▷ Initialise bounds
 7:     else u ← 1
 8:     repeat
 9:         S, Q ← ExploreStateSpace(S, Q)             ▷ (Partially) explore
10:         lower ← BuildLower(S, Q, prop)             ▷ Build MAs
11:         upper ← BuildUpper(S, Q, prop)
12:         l ← Analyse(lower, prop)                   ▷ Analyse MAs
13:         u ← Analyse(upper, prop)
14:     until u − l < ε
15:     return [l, u]
```

and lower bounds (Lines 10 to 11) and analyse both MAs by model checking (Lines 12 to 13) to obtain refined bounds.

We describe the key ingredients of the approximation algorithm in the following. We start by describing the partial state-space generation in Section 6.1.1, then explain the extension and analysis for both *unreliability* and *MTTF* in Section 6.1.2 and finally present three exploration heuristics in Section 6.1.3.

6.1.1 Partial state-space generation

The algorithm for the partial state-space generation is given in Algorithm 2. The function gets the current state space S and an exploration queue Q as input. The algorithm iteratively explores states from the queue Q and adds the newly generated states to S and Q. We stop exploration if either the exploration queue Q is empty—and we have fully explored the state space—or the heuristic stops the exploration, i.e. StopExplore(Q) is true, cf. Line 2. We denote all states which still need to be explored—and are still in the queue Q—as *skipped states* in the following. The function returns the partial state space S and the skipped states Q. The skipped states can be used as input for a new iteration if required.

Note that we obtain the complete state space as presented in Section 5.2.2 if the heuristic never stops the exploration, i.e. StopExplore(Q) always returns false.

Algorithm 2 Partial state-space generation

function ExploreStateSpace(S, Q)
  while $Q \neq \emptyset \wedge \neg$StopExplore$(Q)$ **do**
    $s \leftarrow$ pop(Q) ▹ Get next state from queue
    for $b \in \mathrm{BE}_{\mathcal{F}}$ where b can fail in s **do**
      $s' \leftarrow$ Propagate(s, b) ▹ Get successor state
      if s' is invalid **then continue**
      if $s' \notin S$ **then**
        $S \leftarrow S \cup \{s'\}$ ▹ Add new state
        if s is not failed **then**
          $Q \leftarrow Q \cup \{s'\}$
  return S, Q

Example 6.1 (Partial state space) We use the DFT $\mathcal{F}$ in Figure 6.2 as a running example. The partial state space for $\mathcal{F}$ after one iteration is depicted in Figure 6.3. The skipped states are indicated by grey background colour whereas failed states are indicated by red colour. The iteration only explores the initial state s_0 and adds three new states s_1, s_2 and s_3. State s_2 can already be recognised as failed and is therefore not marked as skipped.

6.1.2 Computing approximation

From the partial state space, we generate MAs for the lower and upper bound. Depending on the property—*unreliability* or *MTTF*—slightly different MAs are constructed.

6.1.2.1 Extended MA for unreliability

The lower bound on the *unreliability* corresponds to the best case in the DFT. We model the best case by setting all skipped states to fail-safe and adding self-loops. As a result, only the currently explored failed states will contribute to the unreliability. For all skipped states, we assume that all future failures will never lead to system failure. Obviously, further exploration of skipped states can only lead to worse results as we might encounter failed states now.

The upper bound on the unreliability correspond to the worst case. We model the worst case by setting all skipped states to failed. Further exploration of skipped states can only lead to better results in this case, because we might encounter states which are not yet failed.

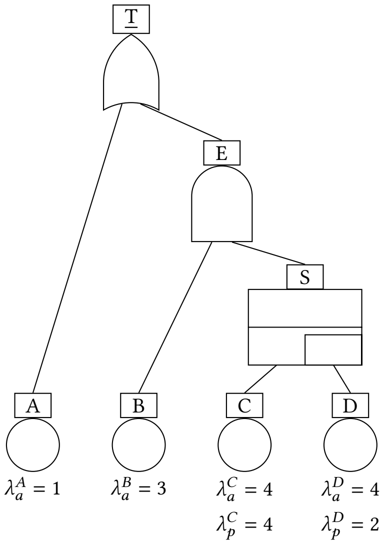


Figure 6.2: Example DFT $\mathcal{F}$ for approximation

Example 6.2 (Extended MA for unreliability) We extend the partial state space of Figure 6.3 for an unreliability property. The resulting MAs for the lower and upper bound are depicted in Figure 6.4.

6.1.2.2 Extended MA for MTTF

Unfortunately, we cannot use the same extended MAs for *MTTF*. The reason is that for the lower bound, all skipped states are made absorbing. Thus, the expected time

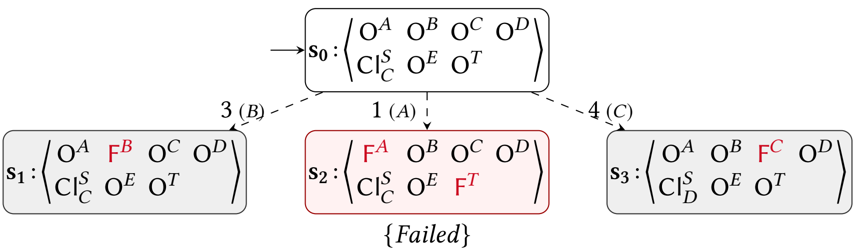
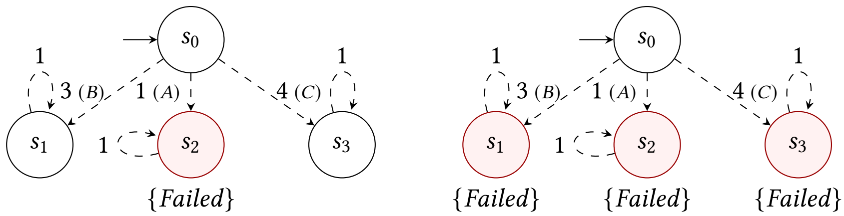


Figure 6.3: Partial state space for $\mathcal{F}$



(a) MA for lower bound

(b) MA for upper bound

Figure 6.4: Extended MAs for unreliability

to reach a failed state from a skipped state is not defined. We can only compute the MTTF if all paths from the initial state eventually reach a failed state. That means the given DFT must eventually fail.

> **Remark 6.1 (Assumption for MTTF approximation)** We assume for the MTTF approximation that the TLE always fails if all BEs failed—except those that cannot fail due to SEQs.

This assumption is only a mild restriction as most realistic systems should exhibit this property. We provide an approach to automatically and efficiently check this assumption in Section 6.2.

The idea behind the extended MAs for MTTF are as for unreliability: we model the worst-case and best-case scenarios in the DFT. Both cases depended on the BEs which are still operational in a state.

operational BEs

Definition 6.1 (Operational BEs) Let state(τ) be a state for DFT $\mathcal{F}$. The *operational BEs* for state(τ) are given as

$$\mathsf{OperationalBEs}(\mathsf{state}(\tau)) = \{v \in \mathsf{BE}_{\mathcal{F}} \mid \mathsf{state}(\tau)_v = \mathsf{O}^v\}\,.$$

Lower bound | The lower bound on the MTTF is given by the worst case behaviour of the DFT. The worst case happens if the next failure of an operational BE leads to complete system failure. This can be thought of as having an OR-gate over the remaining operational BEs. We model this in our MA by adding a transition from each skipped state to a unique failed state s_F. The outgoing rate λ^s_{low} for a skipped state s is given by the sum of the failure rates of all operational BEs:

$$\lambda^s_{\text{low}} = \sum_{v \in \mathsf{OperationalBEs}(s)} \lambda^v_a.$$

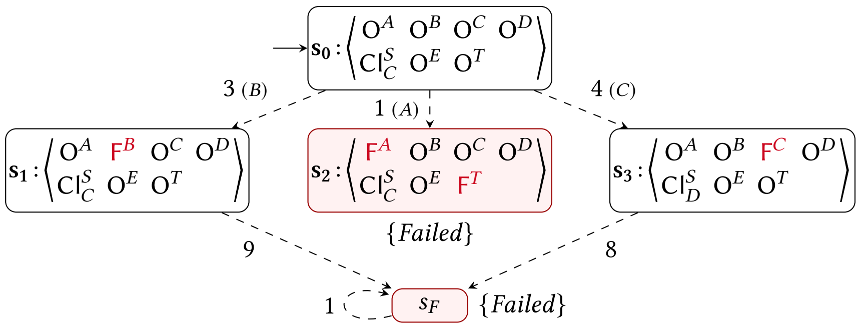


Figure 6.5: MA for lower bound

Note that the rate λ^s_{low} can be refined by using the passive failure rate λ^v_p for inactive elements v instead of the active rate. Furthermore, the rate of *disabled elements*—due to a SEQ—can be ignored as their failure would always yield an *invalid state*. It is important to note that skipped states should not have outgoing *probabilistic transitions*, because otherwise the lower bound would not be sound any more. We therefore always explore states where dependencies can trigger.

Example 6.3 (Lower bound MA for MTTF) We extend the partial state space of Figure 6.3 for MTTF. The resulting MA for the lower bound is depicted in Figure 6.5. We consider for example skipped state s_1. In state s_1, three BEs are still operational: A, C and D. The corresponding failure rate for skipped state s_1 is therefore given by

$$\lambda^{s_1}_{\text{low}} = \lambda^A_a + \lambda^C_a + \lambda^D_a = 1 + 4 + 4 = 9.$$

6

Upper bound | The upper bound on the MTTF is given by the best case behaviour of the DFT. In the best case, the DFT only fails after all remaining operational BEs have failed—it must be failed then due to Remark 6.1. This can be thought of as having an AND-gate over the remaining operational BEs. Unfortunately, the maximum of exponential distributions is not an exponential distribution any more. One possible approach therefore is to generate the complete state space corresponding to the AND-gate. However, this construction is not practicable due to the large number of states, cf. Lemma 5.5 on page 191. Instead, we add a transition from the skipped state to the unique failed state where the transition rate is the reciprocal of the MTTF of the AND-gate. In other words, we need to compute the expected time of the maximum over the failure distributions of the operational BEs.

Lemma 6.1 (Expected time of maximum of exponential distributions) Let D be a set of exponential distributions and let Λ denote the multi-set[1] of rates of distributions in D. We denote the multi-sets of Λ with cardinality i by

$$[\Lambda]^i = \{B \subseteq \Lambda \mid |B| = i\}.$$

·maximum of exponential distributions

The *expected time of the maximum of exponential distributions D* is given by

$$\mathbf{E}\big(\max(D)\big) = \mathbf{E}\big(\max(\Lambda)\big) = \sum_{i=1}^{|\Lambda|} (-1)^{i+1} \cdot \Bigg(\sum_{B \in [\Lambda]^i} \frac{1}{\sum_{\lambda \in B} \lambda} \Bigg).$$

Proof (sketch). The proof is given in Appendix C.3.1 on page 355. ■

In the extended MA, we set the outgoing transition rate λ^s_{up} for a skipped state s as:

$$\lambda^s_{\text{up}} = \frac{1}{\mathbf{E}\big(\max(\Lambda)\big)}, \text{ where } \Lambda = \Big\{ \lambda^v_p \mid v \in \mathsf{OperationalBEs}(s) \Big\}.$$

Note that we can refine the bound by using the active failure rate for elements which are actively used.

In practice, computing the exact value for $\mathbf{E}\big(\max(\Lambda)\big)$ is still expensive and we therefore use an upper bound for faster computation. The goal is to find a good trade-off between the computation time of the upper bound for $\mathbf{E}\big(\max(\Lambda)\big)$ and the tightness of the approximation. For example, truncating the sum by solely considering singleton sets yields an easy to compute upper bound:

$$\mathbf{E}\big(\max(\Lambda)\big) \leq \sum_{i=1}^{1} (-1)^{i+1} \cdot \Bigg(\sum_{B \in [\Lambda]^i} \frac{1}{\sum_{\lambda \in B} \lambda} \Bigg) = \sum_{\lambda \in \Lambda} \frac{1}{\lambda}.$$

Example 6.4 (Upper bound MA for MTTF) We again consider the partial state space of Figure 6.3 . The extended MA for the upper bound is depicted in Figure 6.6. We consider skipped state s_1 where the BEs A, C and D are still operational. The corresponding expected time over the three corresponding failure rates is given by

$$\mathbf{E}\big(\max(\{1,4,2\})\big) = \frac{1}{1} + \frac{1}{4} + \frac{1}{2} - \frac{1}{1+4} - \frac{1}{1+2} - \frac{1}{4+2} + \frac{1}{1+4+2} = \frac{167}{140}.$$

[1] A multi-set is similar to a set but allows multiple entries of the same element.

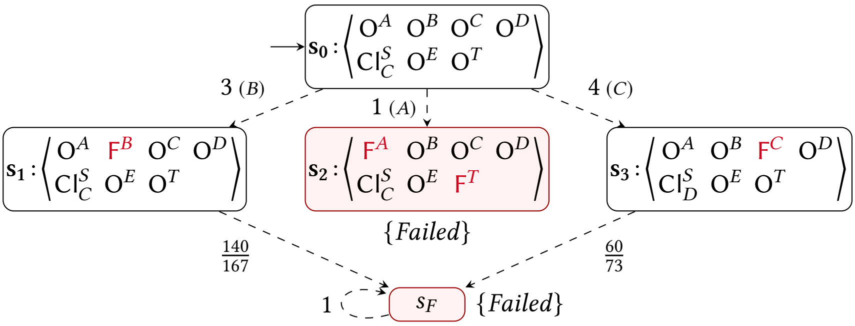


Figure 6.6: MA for upper bound

The outgoing transition rate is therefore $\lambda_{\text{up}}^{s_1} = \frac{140}{167}$. Note that the approximation which solely considers the singleton sets would give an expected time of $\frac{7}{4} = \frac{245}{140}$ which is considerably larger than the exact value.

6.1.3 Exploration heuristics

The quality of the approximation, i.e. how fast we obtain a good error bound for the exact result, is mainly determined by the explored states. We employ an exploration heuristic which has two tasks:

- determining the state exploration order, i.e. which state is taken next from the queue Q in Algorithm 2 on page 222, and
- setting the stopping criteria for the exploration, cf. StopExplore(Q) in Line 2 of Algorithm 2 on page 222.

We present three exploration heuristics in the following.

Depth | The *bounded depth heuristic* explores states up to k consecutive BE failures in iteration k. It is inspired by standards like the ISO 26262 [ISO26262] which requires analysis for two-point failures. The approximation analysis can either stop after a given depth and return sound bounds, or the approach increases the depth until a satisfactory error bound is achieved.

Probability | The *probability heuristic* orders the unexplored states in decreasing order by the probability to reach the state from the initial state. We therefore focus on the most likely failures which carry the most probability mass. During exploration, we continuously update the probability to reach a state if a new ingoing transition is added. We stop the current exploration if either the number of newly explored states is sufficiently large or no state has a probability higher than a given threshold. Afterwards, we perform the analysis and—if necessary—restart the exploration with a smaller probability threshold.

Bound difference | The last heuristic *bound difference* incorporates the knowledge about the previously computed bounds. As a side effect from the model checking, we do not only obtain bounds for the initial state but in fact for each state. We can therefore use the difference between the upper and lower bound for each state as a guiding principle. The idea is that states with a large difference between the bounds profit from further exploration as the bounds will get tighter. On the other hand, if a state has already tight bounds, further exploration will not yield much improvement. For each unexplored state s with bounds $[l, u]$ and probability p_s to reach this state, we set the priority of this state to $p_s \cdot (u - l)$. The exploration therefore focuses on states which have a large bound difference but also a high probability to reach this state. Note that the priorities of the states will be updated after each iteration, because we obtain new bounds through the model checking.

6.1.4 Related work

In general, our approximation method is a form of abstraction refinement [CGL94; CGJL+00]. However, in addition to building an abstraction which over-approximates the behaviour of a system, we also simultaneously construct an under-approximation.

The closest approach to our work is the approximation algorithm for DFTs presented in [Yev16]. The approach basically coincides with our approximation of the *unreliability* when using the *depth* heuristic. The approach in [Yev16] explores states up to a certain number of consecutive failures. For each skipped state s, the probability of being in this state is multiplied by a factor P_s which captures the probability of reaching a failed state from s. For $P_s = 0$, the result is an under-approximation corresponding to our lower bound and for $P_s = 1$, the result corresponds to our upper bound. To obtain a better estimation, the factor P_s can be approximated by analysing an SFT for each state s. This SFT is obtained from the original DFT by eliminating all BEs which are failed in s, changing dynamic gates to AND-gates and considering repeated events as not repeated. As SFT analysis is significantly faster than DFT analysis, the probability can be quickly estimated. However, as already noted by Yevkin: 'there is no guarantee that the truncation method gives upper or lower bound estimation because of the simplifications' [Yev16]. In contrast, the main advantage of our method is that the computed bounds are sound. It is worth mentioning that Yevkin also notes that 'it is enough to calculate the results for truncation orders $T = 2$ and $T = 3$' [Yev16] in order to obtain sufficiently accurate results, i.e. a depth of two or three consecutive BE failures is enough. Once again, our sound approach automatically decides whether the current depth is accurate enough and otherwise explores more consecutive failures.

Approaches based on partial state-space exploration also exist for probabilistic models such as CTMCs [Hav95], Markov decision processes (MDPs) [BCCF+14] and continuous-time Markov decision processes (CTMDPs) [ABHK18]. All approaches iteratively explore parts of the state state space. The first approach [Hav95] uses different heuristics to choose which states to explore next, e.g. breadth-first search or so called maximum "gravity". The breadth-first search corresponds to our heuristic *depth*. The latter two approaches [BCCF+14; ABHK18] use simulation to establish runs in the Markov process. The partial state space is then formed by all states visited during the simulation. The construction is therefore very similar to our approach using the *probability* heuristic. In all three approaches, unexplored states are made absorbing. Over- and under-approximations of the exact result can be computed by marking these states as either goal states or not, While the existing approaches only support computing the *unreliability*, we also provide dedicated handling for *MTTF*. Another difference is that the latter two approaches analyse Markov processes and require dedicated handling for strongly connected components (SCCs) which are absorbing, i.e. can never be left. As the Markov automaton resulting from a DFT is acyclic, such handling is not necessary in our approach.

6.2 SMT encoding of DFT

For the approximation of MTTF, we need to ensure that the TLE eventually fails, cf. Remark 6.1 on page 224. A user might know from domain-specific knowledge that this assumption holds. However, it is preferable to have an automatic check. In the following, we present such a check which is based on a formal encoding of the DFT in *satisfiability modulo theories (SMT)* [KS16b]. More precisely, we encode the DFT into a fragment of linear integer arithmetic called difference logic [CAMN04]. The encoding can be efficiently checked for satisfiability using SMT solvers such as Z3 [MB08]. 6

Idea of SMT encoding | The idea of the SMT encoding is to represent an ftrace by the time of failure of each SE. Then the events resulting from the propagation after an SE failure have the same time of failure as the SE. Each satisfying variable encoding in our SMT encoding therefore represents a valid DFT trace.

Our SMT encoding of a DFT is similar to existing translations into temporal extensions of Boolean algebra [Mer10; EAHT19]. However, instead of the concrete times of failures, we only consider the order in which elements have failed.

> **Definition 6.2 (Variables in SMT encoding)** Let $\mathcal{F}$ be a DFT. For each element $v \in \mathcal{F}$ we introduce a variable x_v with domain $\mathrm{dom}(x_v) = [0, |\mathrm{BE}| + 1] \subseteq \mathbb{N}_0$. For

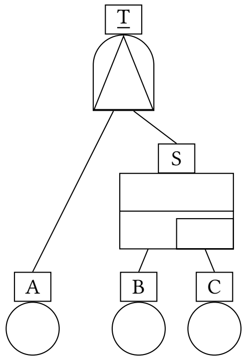


Figure 6.7: Example DFT $\mathcal{F}$ for SMT encoding

each SPARE $s \in \mathrm{SPARE}_{\mathcal{F}}$ and each of its children $v \in \sigma(s)$ we introduce a variable c_v^s with domain $\mathrm{dom}(c_v^s) = [0, |\mathrm{BE}| + 1] \subseteq \mathbb{N}_0$.

A value $x_v \in [1, |\mathrm{BE}|]$ means that DFT element v failed at the same time as the x_v-th BE. Value $x_v = 0$ means that element v was initially failed and value $x_v = |\mathrm{BE}| + 1$ encodes that element v never fails. Note that we extend the original encoding of [6] and also allow $x_v = 0$ to support $\mathrm{CONST}_\top$. For claiming, the value c_v^s indicates the time point when SPARE s claims child v. As before, $c_v^s = |\mathrm{BE}| + 1$ means that the child is never claimed by s.

The SMT variables have a direct correspondence to the events in DFT traces.

·variable assignment

Definition 6.3 (Variable assignment for DFT trace) Let $\mathcal{F}$ be a DFT and let $\boldsymbol{\tau_\mathrm{f}} = \mathbf{f}_{v_1} \ldots \mathbf{f}_{v_n} \in \mathrm{Fails}^{\blacktriangleright}_{\mathrm{SE}_{\mathcal{F}}}$ be an ftrace. The (partial) *variable assignment* $\mathrm{VarAssign}(\boldsymbol{\tau_\mathrm{f}})$ for $\boldsymbol{\tau_\mathrm{f}}$ is given by:

$$x_{v_i} = i \text{ for all } v_i \in \boldsymbol{\tau_\mathrm{f}}.$$

Let $\tau = e_1^0 \ldots e_{m_0}^0\ \mathbf{f}_{v_1}\ e_1^1 \ldots e_{m_1}^1\ \mathbf{f}_{v_2} \ldots \mathbf{f}_{v_n}\ e_1^n \ldots e_{m_n}^n \in \mathrm{Traces}^{\mathit{Storm}}(\boldsymbol{\tau_\mathrm{f}})$ be a trace of $\boldsymbol{\tau_\mathrm{f}}$. The variable assignment $\mathrm{VarAssign}(\boldsymbol{\tau_\mathrm{f}})$ for τ is given by:

$$x_v = \begin{cases} i & \text{if } \exists 1 \leq i \leq n : \mathrm{f}_v \in e_1^i \ldots e_{m_i}^i \\ |\mathrm{BE}| + 1 & \text{otherwise} \end{cases}$$

$$c_v^s = \begin{cases} i & \text{if } \exists 1 \leq i \leq n : \mathrm{cl}_v^s \in e_1^i \ldots e_{m_i}^i \\ |\mathrm{BE}| + 1 & \text{otherwise.} \end{cases}$$

Example 6.5 (Variable assignment in SMT encoding) Let $\mathcal{F}$ be the DFT in Figure 6.7 and let $\tau_{\mathsf{f}} = \mathsf{f}_B\ \mathsf{f}_C\ \mathsf{f}_A$ be an ftrace. The order of failures in the ftrace are represented by the following variable assignment $\mathrm{VarAssign}(\tau_{\mathsf{f}})$:

$$x_B = 1, \qquad x_C = 2, \qquad x_A = 3$$

The trace for τ_{f} is $\tau = \mathrm{cl}_B^S\ \mathsf{f}_B\ \mathrm{cl}_C^S\ \mathsf{f}_C\ \mathsf{f}_S\ \oslash_T\ \mathsf{f}_A\ \oslash_T$. This trace is represented by the following variable assignment:

$$\begin{array}{llll} c_B^S = 0, & x_B = 1, & c_C^S = 1, & \\ x_C = 2, & x_S = 2, & x_A = 3, & x_T = 4. \end{array}$$

The remainder of the encoding describes the failure behaviour of each DFT gate, very similar to the definitions in Chapter 3. We use $n = |\sigma(v)|$ to denote the number of children of a gate.

Definition 6.4 (SMT encoding of DFT gates) DFT gates are encoded as follows:

$$\begin{array}{rl} \mathsf{AND}: & x_v = \max\{x_{v'} \mid v' \in \sigma(v)\} \\ \mathsf{OR}: & x_v = \min\{x_{v'} \mid v' \in \sigma(v)\} \\ \mathsf{PAND}_{\leq}: & x_v = \begin{cases} x_{\sigma(v)_n} & \text{if } \bigwedge_{i=1}^{n-1} x_{\sigma(v)_i} \leq x_{\sigma(v)_{i+1}} \\ |\mathsf{BE}| + 1 & \text{otherwise} \end{cases} \\ \mathsf{POR}_{\leq}: & x_v = \begin{cases} x_{\sigma(v)_1} & \text{if } \bigwedge_{i=2}^{n} x_{\sigma(v)_1} \leq x_{\sigma(v)_i} \\ |\mathsf{BE}| + 1 & \text{otherwise} \end{cases} \\ \mathsf{SEQ}_{\leq}: & \bigwedge\limits_{i=1}^{n-1} x_{\sigma(v)_i} \leq x_{\sigma(v)_{i+1}} \end{array}$$

The VOT_k-gate can be encoded as a combination of AND and OR.

The encoding of the SPARE also needs to encode the claiming with variables c_v^s.

Definition 6.5 (SMT encoding of SPARE) Let $s \in \mathsf{SPARE}_{\mathcal{F}}$ be a SPARE. For a child v of s we denote all other SPAREs which can claim v by

$$\mathrm{SPAREother}_s^v = (\rho(v) \cap \mathsf{SPARE}_{\mathcal{F}}) \setminus \{s\}\,.$$

6

The following constraints encode a SPARE:

$$c^s_{\sigma(s)_1} = 0 \tag{6.1}$$

$$c^s_{\sigma(s)_n} < x_{\sigma(s)_n} \implies x_s = x_{\sigma(s)_n} \tag{6.2}$$

$$\bigwedge_{i=1}^{n-1} c^s_{\sigma(s)_i} < x_{\sigma(s)_i} \iff \psi^s_{i+1}(x_{\sigma(s)_i}) \tag{6.3}$$

$$\psi^s_i(t) = \begin{cases} c^s_{\sigma(s)_i} = t & \text{if } x_{\sigma(s)_i} > t \land \bigwedge_{s' \in \text{SPAREother}_s^{\sigma(s)_i}} c^{s'}_{\sigma(s)_i} > t \\ \psi^s_{i+1}(t) & \text{else if } i+1 \leq n \\ x_s = t & \text{otherwise.} \end{cases} \tag{6.4}$$

Initially, the first child is claimed, cf. Equation (6.1). Equation (6.2) states that if the last child $\sigma(s)_n$ of a SPARE was claimed before it failed, the SPARE fails if this last child fails. For all other children, Equation (6.3) states that if the child was claimed before it failed, then the SPARE tries to claim the next child exactly at the time when the currently claimed child fails. The helper function $\psi^s_i(t)$ encodes that SPARE s tries to claim the i-th child at time t. The definition is given in Equation (6.4). The first case encodes that if the i-th child is not failed at time t and it is also not claimed by any other SPARE at this time, then it will be claimed ($c^s_{\sigma(s)_i} = t$). Otherwise, the SPARE tries to claim the next child ($\psi^s_{i+1}(t)$). If there is no next child, the SPARE fails at this time point ($x_s = t$).

Note that the condition $\bigwedge_{s' \in \text{SPAREother}_s^{\sigma(s)_i}} c^{s'}_{\sigma(s)_i} > t$ in Equation (6.4) is different compared to the original work [6, Equation (7)]. There, we required that the child is never claimed by another SPARE, i.e. $c^{s'}_{\sigma(s)_i} = |\text{BE}| + 1$. This is not correct however[2], because it still allows the case where the child is not claimed yet, because it will be claimed by another SPARE in the future. We fixed this issue by not allowing any 'delayed' claiming. As a side effect, exclusive claiming is now ensured, because a SPARE can only claim a child, if it is available at time t. As a result, the following constraint—which was present in [6, Equation (8)] and ensured exclusive claiming—is not needed any more:

$$\bigwedge_{s \in \text{SPARE}} \bigwedge_{v \in \sigma(s)} \left(c^s_v < |\text{BE}| + 1 \implies \bigwedge_{s' \in \text{SPAREother}_s^v} c^{s'}_v = |\text{BE}| + 1 \right).$$

We can however still include the constraint for better performance of the SMT solver. Similarly, we can explicitly add the constraint that children can only be claimed if they

[2] Thanks to Alexander Bork who found and fixed this issue.

are operational:

$$\bigwedge_{s \in \mathsf{SPARE}} \bigwedge_{v \in \sigma(s)} \left(c_v^s < |\mathsf{BE}| + 1 \implies c_{\sigma(s)_i}^s < x_{\sigma(s)_i} \right)$$

This is also already implicitly encoded in Equation (6.4). Note also that the first child can always be claimed (cf. Equation (6.1)), because it cannot be failed in the beginning, cf. DFT Constraint 9 (No constant failed events in primary modules).

Activation of elements—which is necessary for *cold BEs*—can be encoded by introducing auxiliary variables α_v with domain $\mathrm{dom}(\alpha_v) = [0, |\mathsf{BE}| + 1] \subseteq \mathbb{N}_0$. Variable α_v encodes the time point when element v is activated.

We encode the FDEP behaviour by distinguishing between Markovian failures—where SEs fail on their own—and non-Markovian failures—triggered through an FDEP.

Definition 6.6 (SMT encoding of FDEP) Let $f \in \mathsf{FDEP}_{\mathcal{F}}$ be a dependency. The Boolean variables $\{M_i \mid i \in [0, |\mathsf{BE}|]\}$ encode whether the state visited after i steps is Markovian. The following constraints must hold:

$$M_i \iff \bigwedge_{v \in \mathcal{F}} \left(x_v \leq i \implies \bigwedge_{f \in \mathsf{FDEP}_{\mathcal{F}} \wedge \sigma(f)_1 = v} x_{\sigma(f)_2} \leq i \right) \tag{6.5}$$

$$\neg M_i \implies \bigvee_{f \in \mathsf{FDEP}_{\mathcal{F}}} x_{\sigma(f)_2} = i + 1 \tag{6.6}$$

$$\neg M_i \implies \bigwedge_{b \in \mathsf{SE}_{\mathcal{F}}} \left(x_b = i + 1 \implies \bigvee_{f \in \mathsf{FDEP}_{\mathcal{F}} \wedge \sigma(f)_2 = b} x_{\sigma(f)_1} \leq i \right) \tag{6.7}$$

$$M_i \implies \bigwedge_{b \in \mathsf{SE}_{\mathcal{F}}} (x_b = i + 1 \implies b \text{ not cold in } i + 1) \tag{6.8}$$

A state is Markovian iff all dependent events whose trigger have failed are also failed, cf. Equation (6.5). In other words, all FDEPs which can trigger, have been considered. In a non-Markovian state, the next SE which fails must be a dependent event, cf. Equation (6.6). Moreover, a dependent event can only fail in a non-Markovian state, if the corresponding trigger is already failed, cf. Equation (6.7). Note that in the original encoding [6, Equation (10)], only Equation (6.7) is present but not Equation (6.5). However, the latter constraint is necessary because otherwise the premise $x_b = i + 1$ of Equation (6.7) could always be set to false. Finally, Equation (6.8) expresses that for Markovian states, the failure rate of the next failing SE b must be positive—and b cannot be in cold standby. Formally, we encode this using the activation variable α_b.

Note that we do not support PDEP_p in the SMT encoding. Supporting PDEP_p requires to allow dependent events not to be triggered—because the coin-flip was unsuccessful. That means that in some non-Markovian states, no BE fails—but only the PDEP_p is evaluated—and we obtain 'gaps' in our sequence of BE failures. Moreover, we have to ensure that an unsuccessful PDEP does not trigger its dependent event at a later time point. As PDEPs are rarely used in our examples, we omit them here. Depending on the intended behaviour, one can either replace them by an FDEP in the SMT encoding or remove them.

We have finally all ingredients to define the SMT encoding of a DFT.

·of DFT

Definition 6.7 (SMT encoding of DFT) Let $\mathcal{F}$ be a DFT. The *SMT encoding of* $\mathcal{F}$ is given by formula $\varphi_{\mathcal{F}}$ which is the conjunction of the previous SMT constraints for each DFT element $v \in \mathcal{F}$, cf. Definitions 6.4 to 6.6.

Theorem 6.2 (Correctness of DFT encoding) Let $\mathcal{F}$ be a DFT and $\varphi_{\mathcal{F}}$ be the corresponding SMT encoding. Then $\varphi_{\mathcal{F}}$ encodes the complete behaviour of the DFT $\mathcal{F}$ w.r.t. semantics *Storm*. More formally,

$$\tau \in \mathsf{Traces}^{\mathit{Storm}}(\mathcal{F}) \iff \mathsf{VarAssign}(\tau) \text{ is a satisfying assignment for } \varphi_{\mathcal{F}}.$$

Proof (sketch). The correctness of the constraints for each gate follows by construction as explained before. Moreover, the constraints closely follow the failure conditions of the gates as given in Chapter 3, cf. e.g. Definition 3.8 on page 75. Following Theorem 5.1 on page 185 we know that the propagation in *Storm* always yields a unique state. Thus, a variable assignment $\mathsf{VarAssign}(\boldsymbol{\tau}_\mathsf{f})$ for an ftrace $\boldsymbol{\tau}_\mathsf{f}$ always induces a unique variable assignment $\mathsf{VarAssign}(\tau)$ for the corresponding trace $\tau \in \mathsf{Traces}^{\mathit{Storm}}(\mathcal{F})$. ∎

It remains to encode the sequence of SE failures. In our setting, we are only interested in maximal failure sequences. In other words, we want to find a sequence $\boldsymbol{\tau}_\mathsf{f} \in \mathsf{Fails}^{\blacktriangleright}_{\mathsf{SE}}$ such that any extension of $\boldsymbol{\tau}_\mathsf{f}$ yields an invalid ftrace, i.e. either all SEs are already failed or any additional SE failure violates a SEQ.

We use the Boolean variables $\{y_v \mid v \in \mathsf{SE}_{\mathcal{F}}\}$ to encode whether SE v is part of the sequence $\boldsymbol{\tau}_\mathsf{f}$. Further, we use formula ψ to encode our desired behaviour, e.g. whether the TLE should fail.

·complete SMT encoding

Definition 6.8 (Complete SMT encoding) Let $\mathcal{F}$ be a DFT and $\varphi_{\mathcal{F}}$ the corresponding SMT encoding. Let ψ be a formula encoding the desired behaviour of the DFT element(s). Then $\Phi^{\mathcal{F}}_{\psi}$ is the *complete SMT encoding* for $\mathcal{F}$ and ψ given by the conjunction of the following constraints:

$$\bigwedge_{b \in \mathsf{CONST}_\top} x_b = 0 \tag{6.9}$$

$$\bigwedge_{b \in \mathsf{SE}_\mathcal{F}} y_b \implies \left(1 \leq x_b \leq |\boldsymbol{\tau}_\mathsf{f}| \wedge \bigwedge_{b' \in \boldsymbol{\tau}_\mathsf{f} \setminus \{b\}} x_{b'} \neq x_b\right) \tag{6.10}$$

$$\left(\bigwedge_{b \in \mathsf{SE}_\mathcal{F}} \neg y_b \implies x_b = |\mathsf{BE}| + 1\right) \implies \varphi_\mathcal{F} \wedge \psi \tag{6.11}$$

$$\bigwedge_{b \in \mathsf{BE}_\mathcal{F}} \neg y_b \implies (x_b = |\boldsymbol{\tau}_\mathsf{f}| + 1 \implies \neg\varphi_\mathcal{F}) \tag{6.12}$$

First, we ensure that $\mathsf{CONST}_\top$ fail in the beginning, cf. Equation (6.9). Equation (6.10) ensures that $\boldsymbol{\tau}_\mathsf{f}$ constitutes a totally ordered gap-less sequence. Note that the cardinality $|\boldsymbol{\tau}_\mathsf{f}|$ can be encoded by a linear arithmetic formula over the variables y_v. Equation (6.11) ensures that if only the SEs in $\boldsymbol{\tau}_\mathsf{f}$ fail—and all other SEs never fail—then the desired behaviour ψ occurs in the DFT encoded by $\varphi_\mathcal{F}$. Finally, Equation (6.12) ensures that a failure of one of the SEs not in $\boldsymbol{\tau}_\mathsf{f}$ would always yield invalid behaviour.

Theorem 6.3 (Correctness of SMT encoding) Let $\mathcal{F}$ be a DFT and let $\psi \iff x_{top} = |\mathsf{BE}| + 1$ be the formula encoding that the TLE does not fail. Then

$$\neg\Phi^\mathcal{F}_\psi \text{ holds for all assignments } \iff \mathcal{F} \text{ eventually fails for semantics } \textit{Storm}.$$

Proof (sketch). We prove the equivalent statement

$$\begin{aligned} &\Phi^\mathcal{F}_\psi \text{ is satisfied for an assignment} \\ \iff &\mathcal{F} \text{ does not fail for a maximal trace } \tau \in \mathsf{Traces}^{\textit{Storm}}(\mathcal{F}). \end{aligned}$$

We assume that $\mathcal{F}$ does not fail for maximal trace τ, i.e. $\mathsf{f}_{top} \notin \tau$. With Theorem 6.2 and Definition 6.3 we then know that $x_{top} = |\mathsf{BE}| + 1$. Thus, ψ is satisfied and therefore $\Phi^\mathcal{F}_\psi$ is satisfied.

We assume that $\Phi^\mathcal{F}_\psi$ is satisfied for an assignment. From Theorem 6.2 and Definition 6.3 we know that this variable assignment corresponds to a trace τ by construction, i.e. the assignment is $\mathsf{VarAssign}(\tau)$. As $\Phi^\mathcal{F}_\psi$ is satisfied, ψ is satisfied in particular, i.e. $x_{top} = |\mathsf{BE}| + 1$. That is also reflected in the corresponding trace τ, i.e. $\mathsf{f}_{top} \notin \tau$, and the DFT does not fail for τ.

6

SMT encoding for MTTF assumption | For our partial state-space generation, we have to check whether there is an *ftrace* such that the TLE does not fail, i.e.

$$\psi \iff x_{top} = |\mathsf{BE}| + 1.$$

If there exists a satisfying assignment for the variables such that all constraints of Definition 6.8 are satisfied, then the TLE does not fail in at least one sequence. As a result, Remark 6.1 on page 224 is not correct and we cannot apply the approximation approach for MTTF. We can use the satisfying variable assignments—which encodes the violating sequence—to see whether this behaviour is indeed intended for the modelled system. If however the constraints are unsatisfiable, then we can perform the approximation.

In our experiments, we used the SMT solver Z3 [MB08] and were able to automatically ensure that Remark 6.1 holds for all DFT examples considered in Section 7.2.

SMT encoding for better bounds | The SMT encoding allows for other applications as well. One promising approach is to compute minimal and maximal ftraces. The

·minimal ftrace

minimal ftrace has the fewest number of SE failures s.t. the DFT is failed:

$$min_{\mathcal{F}} = \min_{\tau_f \in \mathsf{Fails}^{\blacktriangleright}_{\mathsf{SE}}} \{ |\tau_f| \mid top \in \mathsf{Failed}(\tau_f) \}.$$

·maximal ftrace

The *maximal ftrace* has the largest number of SE failures s.t. the DFT is still not failed:

$$max_{\mathcal{F}} = \max_{\tau_f \in \mathsf{Fails}^{\blacktriangleright}_{\mathsf{SE}}} \{ |\tau_f| \mid top \notin \mathsf{Failed}(\tau_f) \}.$$

We encode that the TLE fails after exactly i SEs have failed by setting $\psi \iff x_{top} = i$. The minimal ftrace length $min_{\mathcal{F}}$ is given by the minimal i such that the complete SMT encoding is satisfied. The maximal ftrace length $max_{\mathcal{F}}$ is given by the maximal i such that the encoding with $\psi \iff x_{top} = i + 1$ is satisfied.

We can use $min_{\mathcal{F}}$ and $max_{\mathcal{F}}$ to improve the extended MAs for MTTF, cf. Section 6.1.2.2. In particular, we do not consider all operational SEs for the upper bound but only $max_{\mathcal{F}} + 1$ consecutive SE failures, because the TLE is always failed afterwards.

A further improvement would be to dynamically compute the bounds $min_{\mathcal{F}}$ and $max_{\mathcal{F}}$ for each skipped state on the fly. We can encode each failed element v by setting the corresponding SMT variable x_v. Computing the bounds then yields tighter approximations for each skipped state and therefore better bounds overall.

SMT encoding for dependency conflicts | Instead of performing a static analysis to check for *dependency conflicts*, cf. Section 5.4.4, we can use the SMT encoding. We check whether there a two dependencies $f, f' \in \mathsf{FDEP}_{\mathcal{F}}$ whose dependent events can be triggered in arbitrary order. In other words, the trigger of the first dependency fails before or simultaneous to the trigger of the second one, but the dependent event of the first dependency fails after the trigger of the second one fails:

$$x_{\sigma(f)_1} \leq x_{\sigma(f')_1} \wedge x_{\sigma(f)_2} > x_{\sigma(f')_1} \wedge x_{\sigma(f)_1} < x_{\sigma(f)_2} \wedge x_{\sigma(f')_1} < x_{\sigma(f')_2}.$$

The last two terms ensure that the dependent events do not fail on their own before the trigger events. If the constraint is unsatisfiable for two dependencies, they are not in conflict and we can fix an order during state-space exploration instead of introducing a non-deterministic choice.

SMT encoding for minimal cut sequences | The SMT encoding can also be used to enumerate all failure sequences of a given length. This can for example be used to compute all minimal cut sequences. Note though that minimal cut sequences are not adequate for DFT analysis [JGKS16].

Summary

- ➜ Partial state-space generation allows for faster DFT analysis by only generating the most relevant parts of the state space.
- ➜ From the partial state space, Markov automata for the lower and upper bound (of the exact result) are created and analysed by model checking.
- ➜ The approximation returns sound bounds for the exact result which can be iteratively refined by exploring more states.
- ➜ Different heuristics for state exploration can be employed such as only exploring a limited number of consecutive failures or prioritisation by the tightness of current bounds.
- ➜ The behaviour of DFTs can be translated into an SMT encoding. The encoding is used in a pre-processing step to automatically check whether the system eventually fails.
- ➜ The SMT encoding also provides benefits in other context such as improving the MTTF computation or finding dependency conflicts.

7 Implementation and Evaluation

Assumptions

We make the same assumptions as in Chapters 5 and 6. The DFT is *conventional* and we use the inclusive variants $\mathrm{PAND}_{\leq}$, $\mathrm{POR}_{\leq}$ and $\mathrm{SEQ}_{\leq}$.

We implemented both the state-space generation of Chapter 5 and the approximation approach of Chapter 6 in the probabilistic model checker Storm [2; 12]. We present the details of our implementation and accompanying tools in the following. Afterwards, we evaluate the implementation along with all optimisations on the DFT benchmark collection FFORT [RBNS+19]. Moreover, we compare our implementation Storm-dft with the existing tools DFTCalc [ABBG+13] and DFTRES [BRS20], and show that Storm-dft is state-of-the-art for DFT analysis.

7.1 Implementation

7.1.1 Storm-dft

Our DFT analysis is implemented in the probabilistic model checker Storm[1] [2; 12]. Storm-dft as part of Storm is open source, written in C++and available online[2]. Storm comes with a command-line interface and runs natively on Linux and macOS, but can also run on Windows via our Docker containers[3].

Storm | Storm is a modern, modular and efficient model checker for the analysis of systems with inherent uncertainties, i.e. Markov chains or Markov decision processes in both discrete-time and continuous-time variants. In particular, Storm supports *CTMCs* and *Markov automata*. The modular architecture of Storm allows to exchange solvers—e.g. for matrix-vector multiplications, solving linear equation systems, etc.—and thus, offers great flexibility and the easy inclusion of state-of-the-art libraries. Storm is developed since 2012 and encompasses over 170 000 lines of C++ code by now. It has

[1] https://www.stormchecker.org/
[2] https://github.com/moves-rwth/storm/
[3] https://hub.docker.com/r/movesrwth/storm

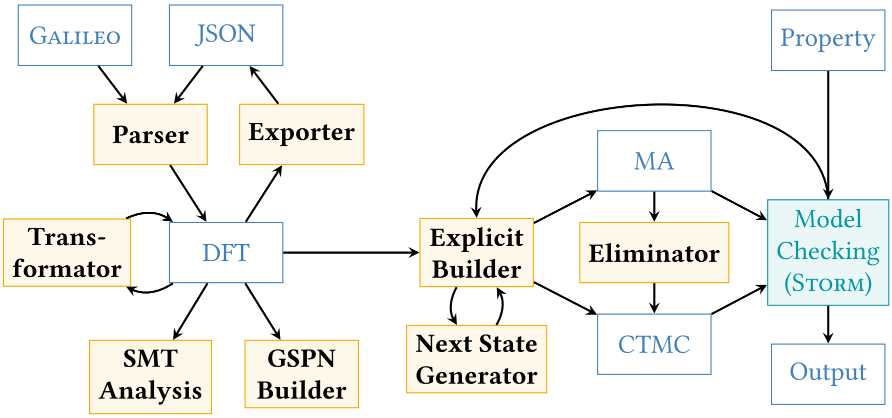


Figure 7.1: Architecture of Storm-dft

reached maturity and is one of the leading probabilistic model checkers as indicated by its performance in the QComp competition[4] [HHHK⁺19; BHKK⁺21].

We use Storm in our DFT analysis as back-end for the analysis of CTMCs and MAs and take advantage of its efficient model checking algorithms as well as its broad spectrum of features.

Architecture of Storm-dft | Storm-dft is implemented as a library and corresponding binary (`storm-dft`) within Storm. The general architecture of Storm-dft is depicted in Figure 7.1. The main data structure is the DFT and its corresponding DFT elements. Each DFT element implements functions which determine how failures, *don't care* and *fail-safe* are propagated. New elements can therefore be added without much implementation effort.

Storm-dft takes a DFT in two input formats:

- the Galileo format[5] [SDC99] introduced by the tool of the same name, and
- our own JSON input format which offers greater flexibility and saves additional information such as positioning information—useful for graphical representations—or the used variants of priority gates.

Storm-dft also provides an Exporter to generate the JSON format from a DFT.

The GSPN Builder generates a GSPN corresponding to the DFT according to the translation in Chapter 3. The resulting GSPN can be exported and used in dedicated GSPN analysis tools such as GreatSPN [ABBD⁺16]. Storm-dft supports SMT Analysis on the DFT as presented in Section 6.2. The resulting SMT formulas can be analysed

[4] https://qcomp.org/

[5] See https://dftbenchmarks.utwente.nl/galileo.html for details

by dedicated SMT solvers such as Z3 [MB08] which is already integrated into Storm. We use the SMT analysis for example as a pre-processing step to determine which dependencies have no *conflict* and can therefore be evaluated in a fixed order.

Before performing the actual analysis, we first have to ensure that the DFT is supported. We use the Transformator to ensure that the resulting DFT contains a unique $\mathrm{CONST}_{\top}$ and only uses binary dependencies. Afterwards, we build the state space for the DFT with the Explicit Builder. This builder iteratively calls the Next State Generator to explore a new state for the state space. The Next State Generator lets a BE fail and performs the necessary failure propagation to obtain a new state. During this process, all enabled optimisations such as *don't care* and *symmetries* are applied. If the complete state space is explored—or a sufficient part when using the approximation—an MA or CTMC is built—depending on whether non-determinism is present in the model or not. If the resulting model is an MA but only contains *spurious* non-determinism, we use the Eliminator to obtain a CTMC, cf. Section 5.4.4. Last, we use Storm to apply Model Checking on the resulting model and user-provided Property. The modular approach of Storm also means that all future improvements within Storm are directly supported in Storm-dft as well. The result of the analysis is provided as Output—or a new iteration with the Explicit Builder is started in case of the approximation.

Technical details | The complete implementation of Storm-dft encompasses more than 11 000 lines of code in C++. All approaches were implemented with a focus on fast computations and the code is therefore heavily optimised—which unfortunately can impact readability. One example for optimisation is the priority queue. The exploration heuristics need to access the next state with the highest priority. However, existing priority queues resulted in too much overhead due to constant resorting. Thus, we implemented our own version of a priority queue based on buckets. As a result, the number of states which can be processed per second increased quite heavily.

Another important point is the usability of Storm-dft. One aspect for example is that DFT elements can not be set to *don't care* if they are *relevant* for the given property, e.g. the property analyses elements other than the TLE. We automatically extract the relevant elements from the given properties and exclude them from certain optimisations.

7

Parametric rates | The complete implementation of Storm-dft makes heavy use of C++ templates. This allows to use different number types for the underlying rates and probabilities. In most cases, we are computing with floating-point numbers. However, Storm-dft also allows to use symbolic *parameters* instead. That way, we can leave open some rates which are possibly not known and determine the MTTF in terms of these parameters. Another possibility is to set a given bound for the resulting

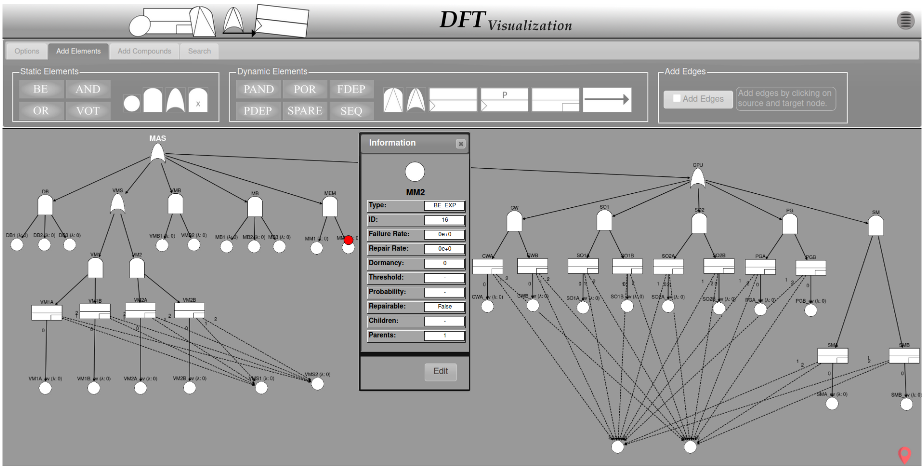


Figure 7.2: Screenshot of DFT GUI

MTTF and synthesise failure rates which ensure this bound. Thus, we can determine how reliable system components need to be in order to guarantee the given safety thresholds. Parameters therefore allow new analysis techniques for the DFT and offer new insights into a given system. We presented an analysis approach for parametric DFTs in [1]. For more details on parameter synthesis in general we refer to [9].

7.1.2 DFT GUI

We also created a Web-GUI for DFTs which is available online[6]. A screenshot of the GUI is given in Figure 7.2. The GUI is based on the JavaScript graph library Cytoscape.js [FLHD+16] and provides an easy-to-use interface for creating DFTs via drag-and-drop. Redundancies can easily be integrated by copying complete subtrees. Further, the layout algorithms provided by Cytoscape.js and the possibility to (temporarily) hide subtrees help retain an overview even for large DFTs. DFTs can be imported and exported in our JSON format which can then be used as input for Storm-dft.

7.1.3 Python bindings

Storm-dft provides a clear workflow: the user inputs a DFT together with a list of properties and Storm-dft computes the desired results. However, in certain scenarios, more flexibility is needed. Thus, we provide Python bindings for Storm-dft in our

[6] https://moves-rwth.github.io/dft-gui/

Python library Stormpy[7]. Using Stormpy allows for rapid prototyping and easy extension of the existing workflow to custom needs. We also developed a (prototypical) Python library for DFT analysis based on Stormpy. This library implements for example most of the *rewrite rules* for simplification of DFTs, cf. Section 5.4.5. Additionally, it implements the criticality computation of DFT elements, cf. Section 5.3.2. Lastly, the library supports the export of DFTs into the Galileo format and as Ti*k*Z pictures[8].

7.2 Evaluation

7.2.1 Set-up

We evaluate our DFT analysis tool Storm-dft on a broad range of DFTs from the literature and compare with existing tools.

FFORT benchmarks | We use the FFORT benchmark collection[9] [RBNS+19] for evaluation. FFORT contains static, dynamic and also repairable fault trees. Each fault tree from the literature can be present in several variants. Some DFTs have scaled-up variants which were created for example by adding redundancies [JGKR+17]. We already evaluated an earlier version of Storm-dft on these scaled DFTs, cf. [1; 6].

We take all fault trees from FFORT except repairable DFTs, because they are not the focus of this evaluation. In total, we have 22 different fault trees in 174 variants. As properties, we compute both *unreliability* and *MTTF*. For unreliability, we use the time bound(s) specified in FFORT and use a default time bound of 1 if none is given. In total, we consider 369 benchmarks.

Configurations of Storm-dft | We use different configurations of Storm-dft in order to asses the influence of the optimisations and approximation heuristics. Concretely, we are using:

- Storm-dft (plain): the baseline of the state-space generation approach without any optimisations.
- Storm-dft (DC): extends Storm-dft (plain) using *don't care*, cf. Section 5.4.1.
- Storm-dft (Symred): extends Storm-dft (DC) additionally employing *symmetry reduction*, cf. Section 5.4.2. This is also the default configuration of Storm-dft and Storm-dft refers to Storm-dft (Symred) in the following.
- Storm-dft (Mod): Storm-dft (Symred) with *modularisation*, cf. Section 5.4.3.

Additionally, we are also evaluating the approximation approach based on partial state-space generation, cf. Section 6.1. We are using Storm-dft (Symred) as base,

[7] https://moves-rwth.github.io/stormpy/

[8] In fact, all DFTs within this work were designed using the DFT GUI and automatically exported to Ti*k*Z.

[9] https://dftbenchmarks.utwente.nl/ffort.php

7

i.e. don't care and symmetry reduction are also enabled for the approximation. If not stated differently, we are using an error bound of $\varepsilon = 10^{-4}$ as termination criterion for the approximation. We use the three exploration heuristics presented in Section 6.1.3:

- Storm-approx (Depth): Storm-dft with the *depth* heuristic.
- Storm-approx (Prob): Storm-dft with the *probability* heuristic.
- Storm-approx (Bound): Storm-dft with the *bound difference* heuristic.

Existing tools | We compare with two existing tools: DFTCalc and DFTRES.

DFTCalc[10] [ABBG+13] is also based on state-space generation, but employs a compositional approach [BCS10]. DFTCalc allows to use different libraries and tools as back-end. We use DFTRES as back-end for DFTCalc in our evaluation as it allows to compute exact values. Moreover, DFTCalc uses Storm as back-end for the model checking. Note that in particular, we do not use CADP [GLMS13]. However, previous work of us already compared DFTCalc using CADP to Storm-dft, cf. [1; 6] The evaluation showed that Storm-dft 'is faster on all benchmarks, and only 11 out of 158 benchmarks could not be analyzed within an hour, whereas DFTCalc could not solve 26 benchmarks' [6].

DFTRES[11] [BRS20] is a rare-event simulator for DFTs, in particular for repairable DFTs. Due to the simulation approach, DFTRES only provides results with a certain confidence. We use a relative error of 10^{-4} in our evaluation. As both other tools are single-threaded, we only use one thread for DFTRES as well. In principle, DFTRES also supports parallelisation which would improve its performance.

Machine info | We run all configurations and tools on a Linux machine with Intel Xeon Platinum 8160 processor with 2.1 GHz. We restrict each benchmark run to 16 GB of memory, a time limit of 1 h and 4 cores. Though all tools run single-threaded, we allow multiple cores as DFTRES is implemented in Java which can use multiple cores.

Artefact | We provide an artefact[12] with the tools, examples and log files. Additional details on the benchmarks and the exact tool configurations are given in Appendix D.

7.2.2 Comparison with existing tools

We start by comparing our implementation Storm-dft with the existing tools DFTCalc and DFTRES. We give the computation times of the different tools and different configurations of Storm-dft in Figure 7.3. The quantile plots depicts how many benchmarks (on the x-axis) could each be solved within the given time (in log-scale on the

[10] https://fmt.ewi.utwente.nl/tools/dftcalc/
[11] https://github.com/utwente-fmt/DFTRES
[12] https://doi.org/10.5281/zenodo.7834056

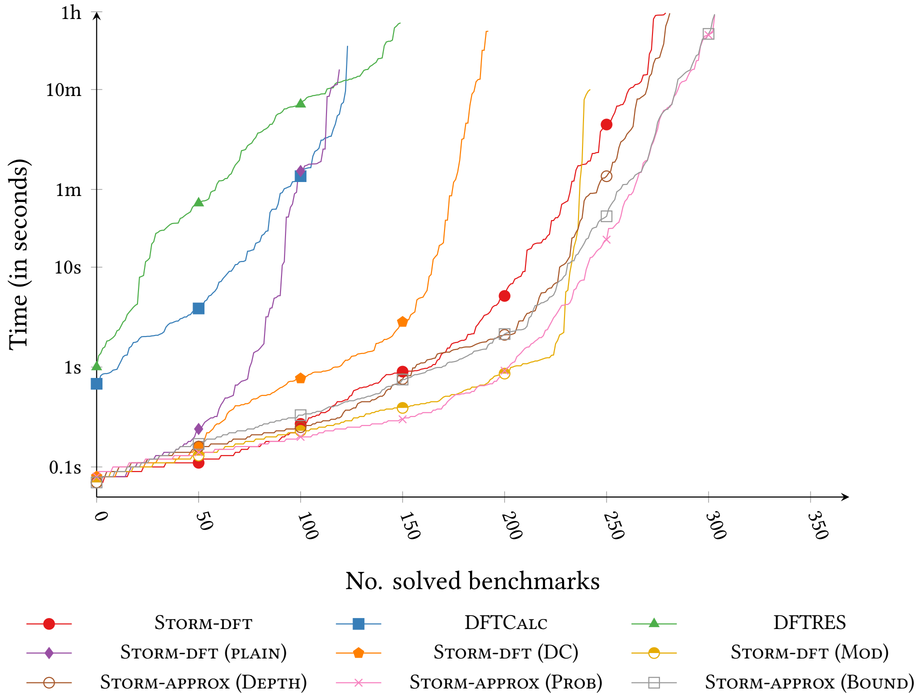


Figure 7.3: Comparison of computation times for all tools and configurations

y-axis). A point (x, y) in the plots means that the x fastest benchmarks could be solved within at most y seconds each. Note that all lines end before the total number of 369 benchmarks, because not all benchmarks could be solved within the given limits.

Computation times | We first compare the default configuration of Storm-dft (red line)—corresponding to Storm-dft (Symred)—to DFTCalc (blue) and DFTRES (green). We clearly see that Storm-dft outperforms both competitors. Due to the fast start-up time of Storm-dft, it can solve 164 benchmarks within 1 s each whereas DFTCalc only solves 11 benchmarks within 1 s and DFTRES one. In fact, Storm-dft solves more benchmarks within 1 s than both DFTCalc and DFTRES in the time limit of 1 h. The high number of quasi instantly solved benchmarks also indicates that at least a third of the considered benchmarks are no challenge for Storm-dft.

Comparing DFTCalc and DFTRES, we see that DFTCalc performs better for smaller benchmarks whereas DFTRES performs better for larger benchmarks and overall solves

Table 7.1: Number of solved benchmarks for different tools and configurations

Tool	# Solved	# TO	# MO	# n.s.	# Err
Storm-dft	280	26	43	14	6
DFTCalc	124	2	191	0	52
DFTRES	150	144	16	11	48
Storm-dft (plain)	120	0	231	14	4
Storm-dft (DC)	193	22	136	14	4
Storm-dft (Symred)	280	26	43	14	6
Storm-dft (Mod)	243	4	0	87	35
Storm-approx (Depth)	282	45	14	14	4
Storm-approx (Prob)	304	38	6	14	7
Storm-approx (Bound)	304	38	6	14	7

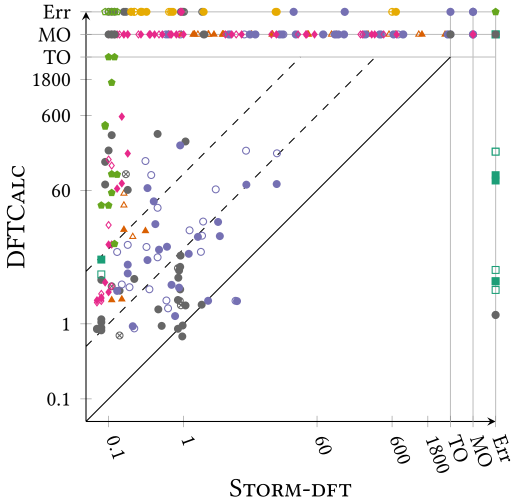
more benchmarks. This is most likely due to the simulation approach of DFTRES which needs to compute a large number of simulation runs even for smaller examples. On the other hand, simulation avoids building the complete state space which is an advantage for larger DFTs. This can also be seen in the slope of the lines for the three tools. The slope of DFTRES is not as steep as the one for DFTCalc and also slightly less steep than the one for Storm-dft. Therefore, DFTRES will most likely perform better when increasing the time limit—but will only provide estimates of the exact result.

Solved benchmarks | We give the exact numbers of solved benchmarks in Table 7.1. For each tool, we give the number of solved benchmarks, the number of benchmarks which timed out (TO) within 1 h, the number of benchmarks which ran out of memory (MO) and required more than 16 GB, the number of benchmarks which are not supported (n.s.) and the number of benchmarks which returned any kind of error (Err).

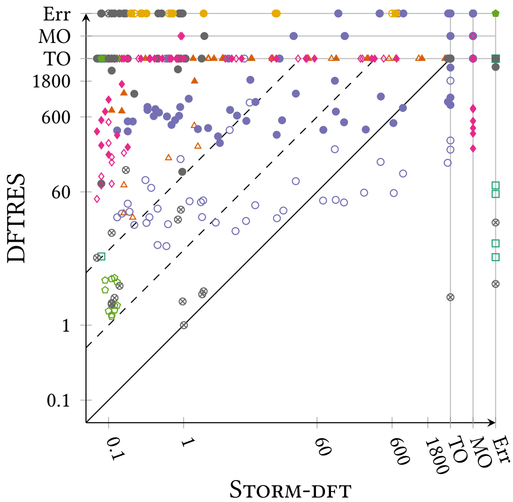
We see that Storm-dft solved roughly more than twice as many benchmarks than DFTCalc and DFTRES. Storm-dft could not solve 25 % of the 369 benchmarks, most of them due to TO or MO. 14 benchmarks are not supported, because they violate DFT Constraint 8 (Independence of modules).

DFTCalc could solve one third of the benchmarks. Most of the unsuccessful benchmarks required too much memory. 52 benchmarks returned an error, but this could also be due to non-supported features of DFTCalc.

DFTRES could solve 40 % of the benchmarks. Most of the unsuccessful benchmarks ran into a TO and a similar number of benchmarks as for DFTCalc encountered an error. This is not surprising as DFTCalc uses DFTRES as back-end.



(a) Computation times compared to DFTCalc (in seconds)



(b) Computation times compared to DFTRES (in seconds)

Figure 7.4: Comparison of Storm-dft with DFTCalc and DFTRES

Detailed comparison of computation times | We give an in-depth comparison between Storm-dft and the other two tools in Figure 7.4. Each scatter plot depicts the computation times both tools need for each benchmark. The time for Storm-dft is given on the x-axis (in log-scale), the time for the other tool is given on the y-axis (also in log scale). Each point (x, y) therefore represents one benchmark that Storm-dft could solve in x seconds and the other tool in y seconds. For better comparison, we added three diagonal lines. Every point above the straight diagonal line means that Storm-dft is faster than the other tool. Every point above the first dashed diagonal line means that Storm-dft is one order of magnitude faster than its competitor and every point above the second diagonal lines represents a faster computation by two orders of magnitude. We also indicate TO, MO and Err. Note that Err in this case combines both n.s. and Err. Dots of the same colour and shape represent different variants of the same fault tree model. Filled dots indicate the property *unreliability* and empty dots indicate property *MTTF*. We distinguish between the six fault trees with the most variants (FTPP, HECS, RC, SF, SMS and VGS) and all other faults trees.

Comparison with DFTCalc | We start by comparing Storm-dft with DFTCalc in Figure 7.4a. We see that nearly all DFTs could be analysed faster by Storm-dft than by DFTCalc. In particular, a large number of benchmarks could be solved by Storm-dft which ran into MO—or Err—for DFTCalc. Interestingly, Storm-dft required anything from under a second up to one hour for the benchmarks which DFTCalc could not solve. This indicates that Storm-dft benefits from the optimisations designed to keep the state space as small as possible whereas DFTCalc runs into a MO. Note that all benchmarks which DFTCalc could solve, also Storm-dft solved. The only exception are most of the FTPP benchmarks, which are not supported by Storm-dft due to violating DFT Constraint 8 (Independence of modules). In total, Storm-dft is at least one order of magnitude faster than DFTCalc for most benchmarks and in a lot of instances even more than two orders of magnitude faster. These results are also consistent with previous comparisons, cf. [1; 6].

Comparison with DFTRES | We compare Storm-dft with DFTRES in Figure 7.4b. Remember that DFTRES only provides estimates of the result with a relative error of 10^{-4} whereas Storm-dft provides accurate results—up to floating point precision. Even though Storm-dft needs to generate the complete state space for a DFT, it is significantly faster than DFTRES on nearly all benchmarks. The notable exception are the fault trees for RC where DFTRES always needs roughly the same time—60 s to compute MTTF and 10 min for unreliability—even for larger variants. On most instances though, Storm-dft is more than two orders of magnitude faster than DFTRES.

In summary, Storm-dft is state-of-the-art for DFT analysis and performs significantly better than its competitors.

7.2.3 Optimisations

The good performance of Storm-dft for DFT analysis stems from the use of several optimisation techniques. We investigate the effect of each optimisation in the following.

7.2.3.1 No optimisations

We start by looking at Storm-dft (plain) which uses no optimisations. From Figure 7.3 on page 245, we see that this configuration of Storm-dft is certainly the worst. But even then, it is on par with DFTCalc. From Table 7.1 on page 246, we see that Storm-dft (plain) solves one third of the benchmarks while the rest almost exclusively run into the memory limit. The state-space explosion problem is very visible here and prevents analysis of larger DFTs.

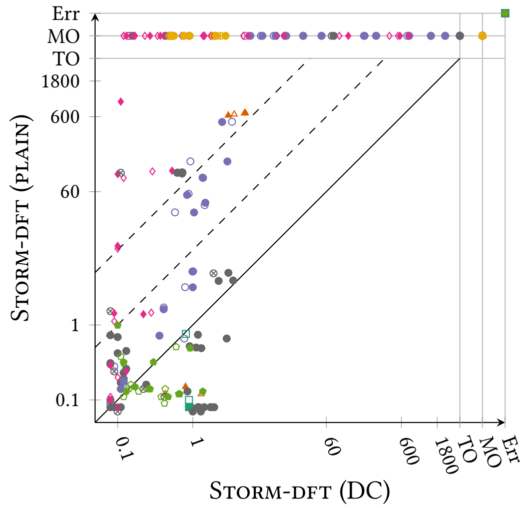
7.2.3.2 Don't care

The first optimisation we look at is the *don't care* (DC) optimisation, cf. Section 5.4.1, which is enabled in Storm-dft (DC). Using DC boosts the performance of the DFT analysis quite significantly, cf. Figure 7.3 on page 245. From Table 7.1 on page 246, we see that 73 additional benchmarks can be analysed with DC compared to Storm-dft (plain). In particular, the number of benchmarks which run into MO is nearly halved.

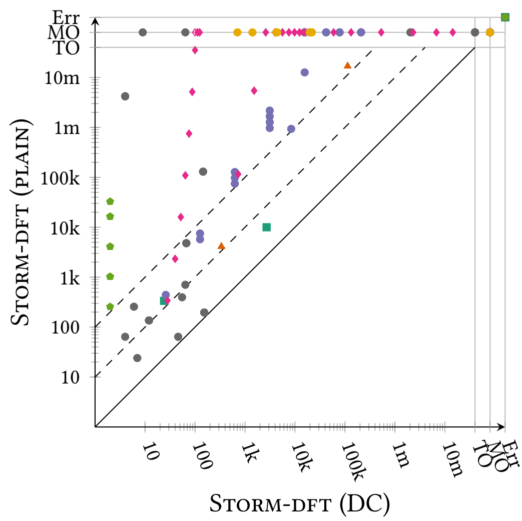
A detailed comparison between using and not using don't cares is given in Figure 7.5. As before, everything above the straight diagonal line indicates that Storm-dft (DC) performs better than Storm-dft (plain). We first look at the computation times of both configurations in Figure 7.5a. For computation times under 1 s, no configuration is clearly better. This stems from the fact that the propagation of DC through the DFT while reducing the state-space size also adds additional overhead. However, for slightly larger DFTs this overhead pays off. All DFTs which Storm-dft (plain) could solve within the time limit are now solved within 10 s using DC. Additionally, a significant portion of previously unsuccessful benchmarks can be solved as well.

The better performance is directly related to the smaller state space which Storm-dft (DC) builds. This can be in Figure 7.5b where we compare the number of states of the resulting Markov chains. Note that the state spaces for both properties unreliability and MTTF are the same and thus, only the filled dots for unreliability are visible. As expected, Storm-dft (DC) never produces larger state spaces than Storm-dft (plain). In fact, for a large portion of the benchmarks, using DC yields state spaces which are more than two orders of magnitude smaller. The advantage of DC is clearly visible for the SF benchmark. While Storm-dft (plain) yields Markov chains which increase by one order of magnitude between the different variants, Storm-dft (DC) yields nearly identical state-space sizes of roughly 100 states.

In summary, using don't cares is crucial to keep the state space small and thereby also decrease computation time.



(a) Computation times (in seconds)



(b) State space sizes

FTPP HECS RC SF SMS VGS Other

Figure 7.5: Influence of using don't care (DC)

7.2.3.3 Symmetry reduction

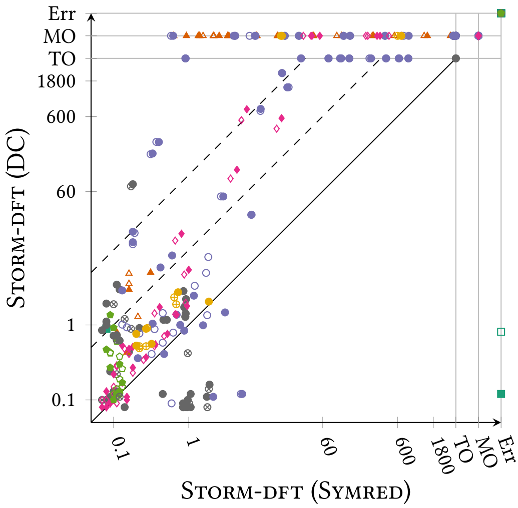
Next, we evaluate the influence of *symmetry reduction*, cf. Section 5.4.2, which is enabled in Storm-dft (Symred). Note that Storm-dft (Symred) also uses don't cares.

Exploiting symmetries again improves the performance of the DFT analysis compared to the previous configuration. This is visible in Figure 7.3 on page 245 where Storm-dft represents Storm-dft (Symred). Note in particular that the slope of the graph for Storm-dft is more flat than the one for Storm-dft (DC). This indicates that the symmetry reduction works well for more complex DFTs. From Table 7.1 on page 246, we see that 87 benchmarks can be solved more using symmetries compared to Storm-dft (DC). In particular, the number of benchmarks which run into the memory limit is reduced by two thirds. Note however, that the structure of the larger benchmarks highly benefits Storm-dft (Symred). Most larger variants are scaled up by adding redundancies which are perfect for symmetry reduction.

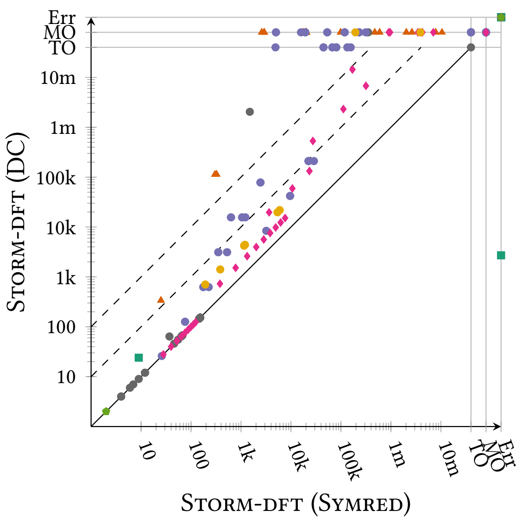
A detailed evaluation of the symmetry reduction is presented in Figure 7.6. We compare Storm-dft (Symred) which exploits symmetries with Storm-dft (DC). A comparison of the computation times for both configurations is shown in Figure 7.6a. For smaller DFTs with an analysis time under 10 s, the overhead introduced by computing the symmetries is greater than its benefit in some instances. However, for larger DFTs, the computation time with symmetry reduction is always lower compared to Storm-dft (DC). In particular, the scaled variants of HECS, RC and SF can be solved significantly faster—or even solved at all—when using symmetries.

The corresponding state-space sizes are depicted in Figure 7.6b. We see that for growing state-space sizes, the benefit of symmetry reduction increases as well and can lead to one order of magnitude fewer states. Also note that we are able to generate and analyse state-space sizes of nearly 10 million states within the time limit of 1 h. Most of the smaller benchmarks do not contain any symmetries which can be exploited. However, this is not a problem, as most of them have 100 states or less anyway. Note in particular that apart from a few instances, all 'other' benchmarks—which do not belong to one of the six larger families of benchmarks—yield Markov chains with less than 100 states. This indicates that the FFORT benchmark collection mostly contains small DFTs—or at least DFTs resulting in small state spaces. Thus, it would certainly be helpful for tool development if larger—industry-sized—DFTs would be added. To this end, we present large DFTs resulting from two industrial case studies in Chapters 8 and 9.

In summary, exploiting symmetries—if present—leads to smaller state spaces and brings DFTs into reach which could not be analysed before.



(a) Computation times (in seconds)



(b) State space sizes

FTPP HECS RC SF SMS VGS Other

Figure 7.6: Influence of symmetry reduction

7.2.3.4 Modularisation

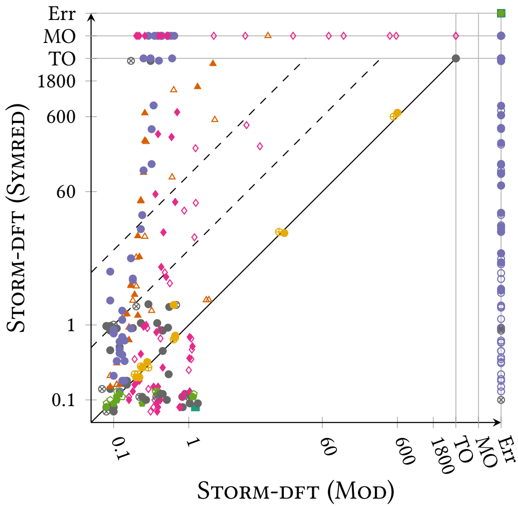
Last, we look at *modularisation*, cf. Section 5.4.3, which is enabled in Storm-dft (Mod). Note that Storm-dft (Mod) also uses both don't cares and symmetry reduction.

As already stated in Section 5.4.3, modularisation works best for a *probability property* as we only need to create the Markov automata for each *independent subtree*. For an *expected-time property* in contrast, we still still have to build the complete state space by constructing the Cartesian product of the independent subtrees. We can glimpse the difference between the two types of properties in Figure 7.3 on page 245. For the first 220 benchmarks, Storm-dft (Mod) performs best and analyses each of the benchmarks within 1 s at most. After that though, only a handful more of benchmarks are solved and the performance in the end is worse than without modularisation, cf. Storm-dft.

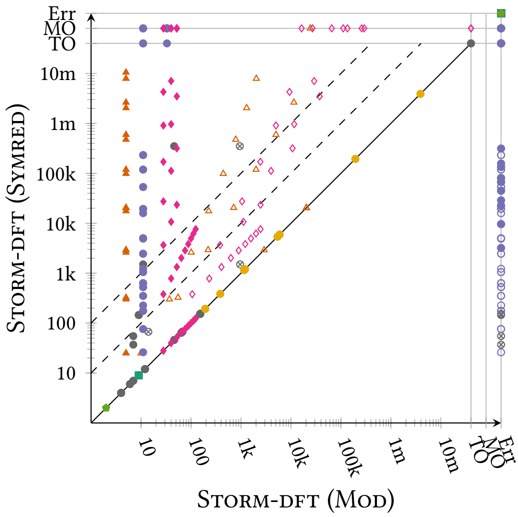
This can also be seen in the number of solved benchmarks as given in Table 7.1 on page 246. In contrast to Storm-dft (Symred), 37 benchmarks less can be solved while using modularisation. Interestingly, this is not due to TO or MO, but mostly due to missing support. The issue here is that the Cartesian product is currently only implemented for CTMCs but not for MAs. Moreover, bisimulation minimisation for MAs with a sparse matrix representation is also not supported yet. For RC, most of the 68 benchmarks result in MAs which can therefore not be analysed for MTTF. The Err stem mostly from RC as well. While computing the unreliability, Storm itself warns that the underlying model checking might produce inaccurate results due to numerical issues. While the resulting values coincide with the results from the other configurations, we still mark them as errors here, because without such comparison we could not trust these values. Putting aside these two issues, we see that modularisation solves all benchmarks except four. Thus, it is very promising to further extend the support of modularisation—also for MAs.

We compare the configurations with and without modularisation in Figure 7.7. From the timings in Figure 7.7a, we first see that only the RC benchmarks could not be handled—as explained before. Apart from that, a large portion of the benchmarks could be solved within 1 s. We clearly see that modularisation is very fast—if applicable. There are some examples—namely VGS—which do not have independent subtrees and therefore do not profit from modularisation.

We depict the state-space sizes in Figure 7.7b. As modularisation produces multiple DFTs, we give the maximum number of states over all subtrees for Storm-dft (Mod). In the plot, the scaled variants of different DFT examples are clearly visible. For HECS, RC and SF, most variants have the same maximal number of states when using modularisation. For a large number of benchmarks, less than 100 states need to be generated for each subtree. Even if larger state spaces need to be build, Storm-dft (Mod) requires one order of magnitude less states than Storm-dft (Symred). From the plot, we also see the different approaches for unreliability and MTTF. The unreliability can be computed on each subtree and only small state spaces are generated. In contrast,



(a) Computation times (in seconds)



(b) State space sizes

FTPP HECS RC SF SMS VGS Other

Figure 7.7: Influence of modularisation

the computation of MTTF still needs to create one large CTMC by building the Cartesian product of all smaller CTMCs. Nevertheless, the resulting CTMC is often several times smaller than the state space without modularisation, because we apply bisimulation minimisation [BK08] on the smaller CTMCs before performing the composition.

In summary, modularisation—if applicable—requires only a fraction of the state space compared to the complete DFT and can compute the unreliability within seconds.

7.2.4 Other models

We shortly take a look at other DFTs not contained in the FFORT benchmark collection.

'Hard' examples for state-space-based analysis | The fictitious example in [BD05b] is meant to be a hard problem for Markov chain-based analysis approaches:

> 'The sole purpose of this example is to illustrate the problem encountered in solving the DFT by generating the MC [...]. [T]he presence of two cascaded PANDs (one of which is the top gate) and the relatively large number of basic components make the corresponding Markov chain of this tree drastically large and practically impossible to solve without resorting to simplifying assumptions and/or approximations (such as state truncation).' ([BD05b])

We solved this model in 0.013 s with a state space of only 514 states. The exact unreliability is 2.016×10^{-10} whereas the original (approximated) result was 1.9×10^{-10} [BD05b].

For the highly-coupled DFT in [GLYZ+15a], the authors stated that 'it is very difficult to obtain the occurrence probability of the top event by the Markov method' [GLYZ+15a]. We obtained a result within 0.01 s with a state space of 234 states. Note that the unreliability values given in [GLYZ+15a] for smaller time bounds are three times higher than the values computed by Storm-dft. For larger time bounds, the values of both approaches get closer.

Nuclear power plant | An emergency power supply of a nuclear power plant was modelled as a *Boolean-logic driven Markov decision process (BDMP)* and presented as a challenge in [Bou17]. We translated the BDMP into a DFT as presented in [10]. The resulting DFT consists of 200 elements and leads to a Markov automaton with 1.5 million states. The MA cannot be checked with Storm within 1 h. However, we know by construction that there are no *dependency conflicts*. We therefore automatically fixed an order for the dependencies and then Storm-dft analysed the DFT in 14 s and built a CTMC of only 16 386 states.

Other examples | We computed unreliability values for the DFT in [YY08] in 0.025 s. Interestingly, our results agree with the results of Galileo—which are given in the paper—but not with the values of the proposed approach and a Monte Carlo simulation.

The two examples in [ADH03] could be analysed within less than 0.015 s. For the second example, our computed unreliability is smaller.

The spare system and the power converter from [ZZDS01] could be analysed within 0.002 s and 0.045 s, respectively. We obtained the same values as the original approach.

In summary, other DFTs from the literature do not pose a challenge for Storm-dft.

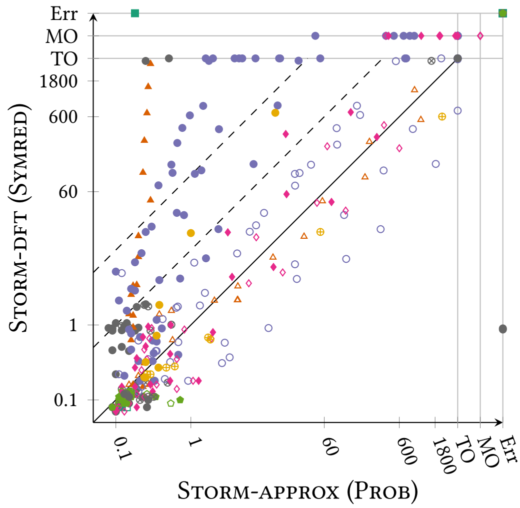
7.2.5 Approximation

We evaluate the approximation approach presented in Section 6.1 and in particular the three different exploration heuristics introduced in Section 6.1.3.

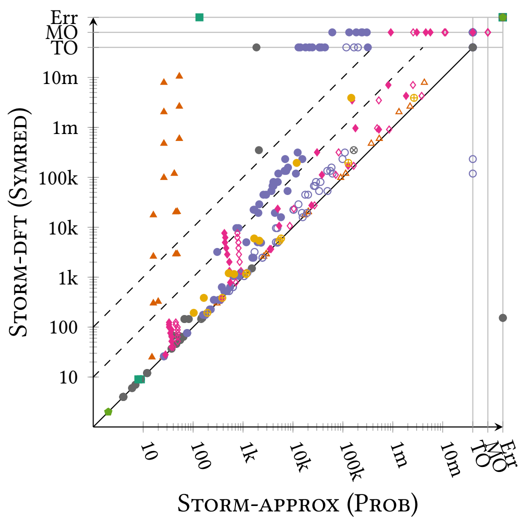
Overview of results | We consider again the quantile plot in Figure 7.3 on page 245. We see that all three approximation approaches perform better than the exact method Storm-dft. Heuristic Storm-approx (Prob) performs best for the first 200 benchmarks and is even on par with the modularisation. While the performance of Storm-approx (Depth) is similar to Storm-approx (Bound) for the fastest 220 benchmarks, Storm-approx (Bound) performs better for the larger benchmarks. For the largest benchmarks, no difference can be observed between Storm-approx (Prob) and Storm-approx (Bound). This also indicates that the additional information of the computed bounds—as used in Storm-approx (Bound)—does not significantly change the state-space exploration. The same can be seen from Table 7.1 on page 246, where no difference between both heuristics Storm-approx (Prob) and Storm-approx (Bound) is visible. In contrast to the original approach Storm-dft (Symred), 24 additional benchmarks can be solved with the approximation approach. In particular, the approximation prevents several MO but has more TO. That means that running the approximation approach for a longer time limit might yield results, because the resulting state space is still manageable. Also note that the approximation provides intermediate bounds for the exact result. That means even the unsolved benchmarks in our examples still provided bounds for the result—though not precise enough.

Detailed comparison of computation times | We give a detailed comparison between Storm-approx (Prob) and Storm-dft in Figure 7.8. We chose the *probability* heuristic, because it performed better than the *depth* heuristic. Moreover, there was no clear difference to the *bound difference* heuristic on the larger DFTs and Storm-approx (Prob) performed better on the smaller DFTs.

We depict the computation times of Storm-approx (Prob) and Storm-dft in Figure 7.8a. For some benchmarks, the exact approach performs better. This is mostly in



(a) Computation times (in seconds)



(b) State space sizes

FTPP HECS RC SF SMS VGS Other

Figure 7.8: Comparison of Storm-approx (Prob) and Storm-dft

cases where nearly the complete state space needs to be explored by the approximation. In these instances, the overhead from building MAs in each iteration and model checking them is too large. Additionally, nearly all such instances compute the MTTF. That means the bounds do not converge fast enough to the exact result for MTTF and tighter upper and lower bounds are needed. A possible improvement is to use additional knowledge from the SMT encoding, cf. Section 6.2, or to use knowledge of the structure of the DFT during the partial state-space generation, e.g. in the heuristic which states to expand.

Overall, the approximation performs better than the exact approach. For some instances, the approximation requires one or more orders of magnitude less time. In particular, several benchmarks which ran into TO or MO before, can now be solved.

We depict the resulting state-space sizes in Figure 7.8b. As expected, the approximation always provides smaller or equal state spaces. In most instances, the partial state space is not significantly smaller than the complete state space, but nonetheless, every state-space reduction yields lower computation times. For some benchmarks, the approximation yields up to 10 times smaller state spaces, e.g. HECS, RC or SF. HECS in particular profits from the partial state-space generation as the required precision can in most cases be computed with a state space of less than 100 states. This also indicates that large parts of the HECS DFTs have nearly no influence on the system unreliability.

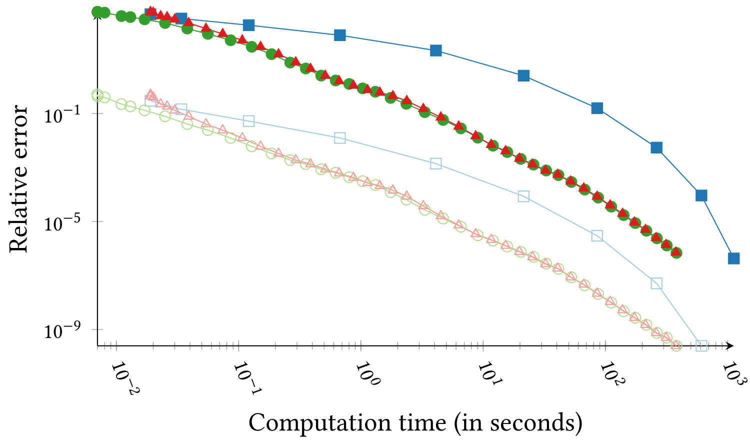
Detailed look at RC example | We take a detailed look into the behaviour of the approximation for an exemplary DFT. We use RC benchmark variant `rc.2-4.hc` with error bound 10^{-6} for presentation purpose here, but the insights also hold in general.

We depict the relative error of the bounds over time in Figure 7.9. Note that both axes are in log-scale. For each of the three heuristics, we depict the errors of the upper and lower bound, where the lower bound is depicted in lighter shade. Each dot represents the result after one iteration.

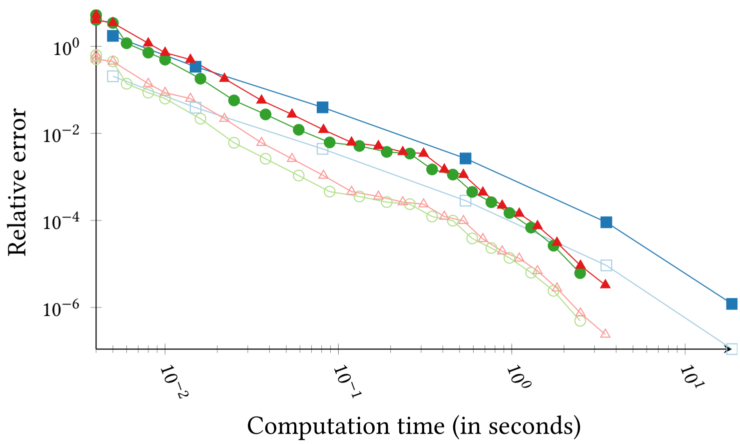
The results for property *MTTF* are given in Figure 7.9a. We see that the relative error of the lower bound is several orders of magnitude lower than the error of the upper bound. This indicates that the worst-case scenario—as assumed for the lower bound—is closer to the actual result than the best-case scenario—as assumed for the upper bound. The lower bound satisfies a given error bound significantly earlier that the upper bound and the remainder of the computation time is then spend on tightening the upper bound. Thus, it is advisable to further improve the computation of the upper bound in order to increase performance of the approximation approach. As stated before, a possible improvement is to use the SMT encoding, cf. Section 6.2.

The results for property *unreliability* are depicted in Figure 7.9b. Here, the difference between upper and lower bounds is not as large as for MTTF. As a result, the computation time is significantly less than for MTTF.

For both properties, we see that Storm-approx (Depth) performs worse than



(a) Property MTTF



(b) Property unreliability

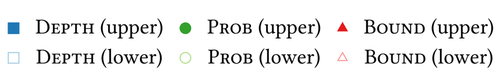


Figure 7.9: Comparison of approximation heuristics on RC

7

the other two heuristics. Using only the number of consecutive failures as guiding principle is therefore not the best approximation approach. This is in particular interesting as other approximation approaches such as [Yev16] are based on limiting the number of BE failures. Storm-approx (Prob) and Storm-approx (Bound) are nearly indistinguishable. This indicates that the additional knowledge from the bounds in Storm-approx (Bound) does not contribute much to the exploration priorities.

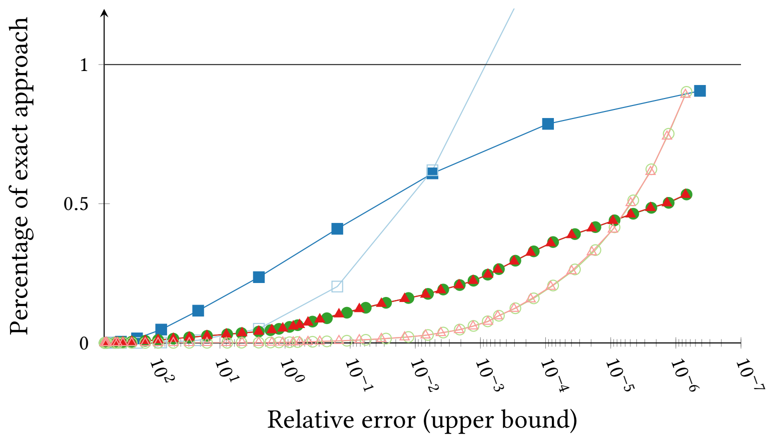
One advantage of the approximation is that it provides reasonable estimates of the exact result in a very short time already. After 10 s, the relative error for MTTF is below 10^{-1} for the upper bound and even below 10^{-5} for the lower bound. Similarly, the relative error for the unreliability is below 10^{-2} for both bounds after only 0.1 s. However, tightening the bounds becomes computationally more expensive for smaller error bounds. For example, computing the lower bound with an error of 10^{-5} takes under 7 s. However, pushing this error bound to 10^{-6} takes additional 13 s.

Comparison with exact approach | Lastly, we exemplary compare the behaviour of the approximation to the exact approach in Figure 7.10. We compare the computation of MTTF on two different benchmarks: RC benchmark variant `rc.2-4.hc` from before (in Figure 7.10a) and the largest VGS variant `vgs-8` (in Figure 7.10b). For each of the three heuristics, we give the number of states and the computation time required to compute a certain error bound. We depict the relative error bound of the upper bound on the (logarithmic) x-axis. Note that the direction is reversed and the error bound decreases to the right. On the y-axis, we give the percentages of the number of states and computation time (in lighter shade) in comparison to the exact approach. For example, a point (x, y) for the computation time represents that for a relative error of x, the approximation requires $y\%$ of the total computation time of the exact approach.

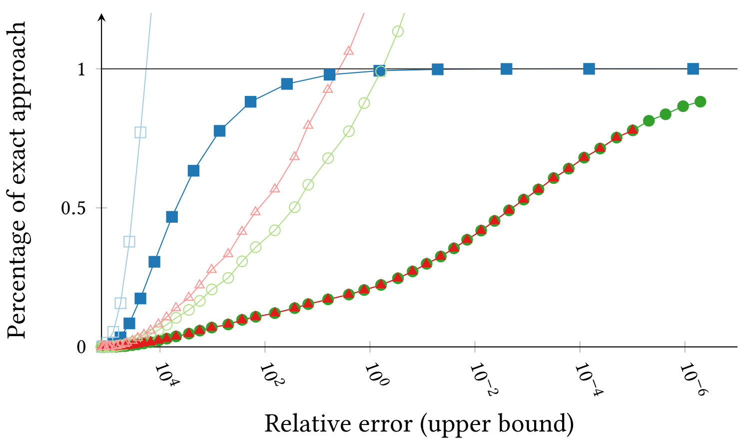
As before, we see that Storm-approx (Depth) performs worse than the other heuristics. The heuristic requires more computation time than the exact approach even for reasonable error bounds. The main reason is that this heuristic explores far more states for the same error bound than the other two heuristics. Thus, performing approximation based on the number of BE failures does not perform well.

We see the strength of the approximation for the other two heuristics: fewer states than for the exact approach need to be explored to obtain tight bounds. For RC, it suffices to build 50 % of the complete state space to obtain bounds with an error of less than 10^{-5}. For VGS, half of the complete state space gives an error bound of 10^{-2}. Note that we are considering the upper bound of the MTTF here, i.e. the worst-case behaviour of the approximation. Results for unreliability for example will be better.

One major problem with the approximation is the additional overhead introduced by building two MAs and model checking them in each iteration. The effect can be seen for VGS, where the computation time exceeds the computation of the exact approach (520 s) already for an error of 1. The overhead could be reduced by performing fewer



(a) Benchmark RC



(b) Benchmark VGS

Figure 7.10: Comparison of states and computation time to exact approach

iterations, i.e. exploring more states in one iteration. This requires careful evaluation however, because we would like to use as few iterations as possible to obtain a result very close to the desired error bound. On the other hand, if the error bound is not yet reached, an additional—then longer—iteration requires far more computation time.

In summary, the approximation outperforms the exact approach for large DFTs and reasonable error bounds, e.g. 10^{-4}. The approximation still leaves room for improvements which could boost the performance even further, e.g. tighter bounds for the upper bounds—especially for MTTF—or decreasing the computational overhead by performing fewer iterations.

Summary

- The DFT analysis and the approximation is implemented as Storm-dft in the probabilistic model checker Storm. Storm-dft is open-source and available at stormchecker.org.
- Alongside the command line interface of Storm-dft, there also exists a graphical web-editor for DFTs and Python bindings for Storm-dft.
- The evaluation shows that Storm-dft is state-of-the-art for state-space-based DFT analysis.
- Storm-dft performs significantly better than the existing tools DFTCalc and DFTRES: it can analyse more DFTs from the FFORT benchmark collection and is faster overall.
- All optimisations—don't cares, symmetry reduction and modularisation—yield significant performance improvements for the DFT analysis.
- Don't care optimisation is crucial to keep the state spaces smaller and significantly improves the performance of the DFT analysis.
- Symmetry reduction can reduce the resulting state space and thereby the analysis time if symmetries are present.
- Modularisation can compute the unreliability within seconds if independent subtrees are present. Computation of MTTF via modularisation could still be improved.
- The approximation approach based on partial state-space generation outperforms the exact approach. Its main benefits are obtaining good error bounds already within the first couple of seconds and exploring less states. For very tight error bounds however, nearly the complete state space needs to be explored.
- Exploration heuristics *probability* and *bound difference* perform similarly for the approximation while heuristic *depth* is significantly worse.
- The approximation leaves room for future improvements, e.g. tighter upper bounds.

Part IV

Applications

8 Vehicle Guidance System

Assumptions

The DFT is *conventional* and uses inclusive variants $\text{PAND}_{\leq}$, $\text{POR}_{\leq}$ and $\text{SEQ}_{\leq}$.

In the third part of this work, we apply our DFT analysis tool to two case studies. We begin with a case study from the automotive domain: a vehicle guidance system.

The content of this chapter was presented in 'Safety analysis for vehicle guidance systems with dynamic fault trees' [8] and the previous conference paper 'Model-Based Safety Analysis for Vehicle Guidance Systems' [3]. Large parts of this chapter are taken verbatim from our publication [8].

8.1 Introduction

Motivation | Cars are nowadays equipped with functions, often realised in software, to e.g. improve driving comfort and driving assistance (with a tendency towards autonomous driving). These functions impose high demands on the required functional safety. ISO 26262 [ISO26262] is the basic norm for developing safety-critical functions in the automotive setting. It enables car manufacturers to develop safety-critical devices—in the sense that malfunctioning can harm persons—according to an agreed technical state-of-the-art. The safety-criticality is technically measured in terms of the so-called *Automotive Safety Integrity Level (ASIL)*. This level takes into account driving situations, failure occurrence, the possible resulting physical harm, and the controllability of the malfunctioning by the driver. The result is classified from QM (no special safety measures required) over ASIL A, B and C up to the most stringent level ASIL D. To meet the functional safety requirements, it is crucial to execute the software functions with a sufficiently low probability of undetected dangerous hardware failures. We consider the design-phase safety analysis of the vehicle guidance system, a key functional block of a vehicle with a high safety integrity level ASIL D, i.e. allowing not more than 10^{-8} residual hardware failures per hour. The key points of our approach are: Automotive Safety Integrity Level (ASIL)

1. manually constructing dynamic fault trees from industrial system descriptions and combining them—in an automated manner—with hardware failure models for several partitionings of functions on hardware, and
2. analysing the resulting overall DFTs by means of probabilistic model checking.

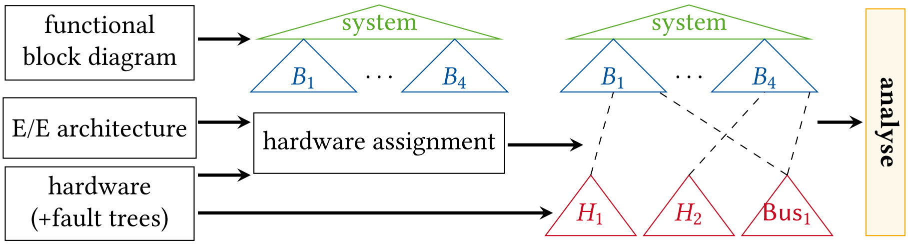


Figure 8.1: Overview of the model-based safety approach

A model-based approach | We summarise the approach in Figure 8.1. The details are discussed in Section 8.3. The failure behaviour of the functional architecture, given as a *functional block diagram (FBD)*, is expressed as a two-level DFT. The upper level of the DFT (green colour) models a system failure in terms of block failures B_i while the lower level (blue colour) models the causes of block failures B_i. The construction of the DFTs is presented in Section 8.3.1. The use of fault trees is natural: they are a well-known model in reliability engineering and thus, no familiarity with additional formalisms is required. Fault trees for hardware components are typically provided by manufacturers. Failures in function blocks can also easily be described by fault trees. The use of DFTs rather than static fault trees allows to model warm and cold redundancies, spare components, and state-dependent faults.

Each functional block is assigned to a hardware platform according to the given E/E architecture and used hardware, cf. Section 8.3.2.2.

The failure behaviour of the hardware platforms are modelled by DFTs H_i (red colour). Depending on the partitioning, the communication goes via different fallible buses that are also modelled by DFTs Bus_i (red colour). The details are given in Section 8.3.2.1.

From the partitioning—and the DFTs of the hardware and the functional level—an overall DFT is constructed in a fully automated manner, cf. Section 8.3.2.3. The overall DFT consists of three layers: (1) *system layer*, (2) *block layer*, and (3) *hardware layer*.

Analysis | We exploit *probabilistic model checking (PMC)* to analyse the DFT of the overall vehicle guidance system. PMC can be used as a black-box algorithm—no expertise in PMC is needed to understand its outcomes—and supports various metrics that go beyond reliability and MTTF, cf. Section 5.3. While they are all expressible by a combination of PMC queries, the number of queries is prohibitively large for some measures relevant for the safety analysis of highly automated cars. Thus, we develop dedicated algorithms to compute these measures within the probabilistic model checker Storm [12], by reusing building blocks for standard PMC queries. In contrast to simulation and statistical model checking [LDB10; AP18], where results are obtained with a given statistical confidence, PMC provides hard guarantees that the safety objectives are met. These guarantees are important as ISO 26262 requires that 'metrics are verifiable and precise enough to differentiate between different architectures' [ISO26262, 5:8.2].

Whereas most ISO 26262-based analyses focus on single and dual-point failures, PMC naturally supports the analysis of multi-point failures of the vehicle guidance system's DFT. Consideration of multi-point failures is highly relevant, as 'it is necessary to consider multiple-point failures of a higher order than two in the analysis when the technical safety concept is based on redundant safety mechanisms' [ISO26262, 5:9.4.3.2]. To limit the computation time, we use the approximative computation of Storm-dft (cf. Chapter 6): instead of a precise value, we compute sound upper and lower bounds for the measures.

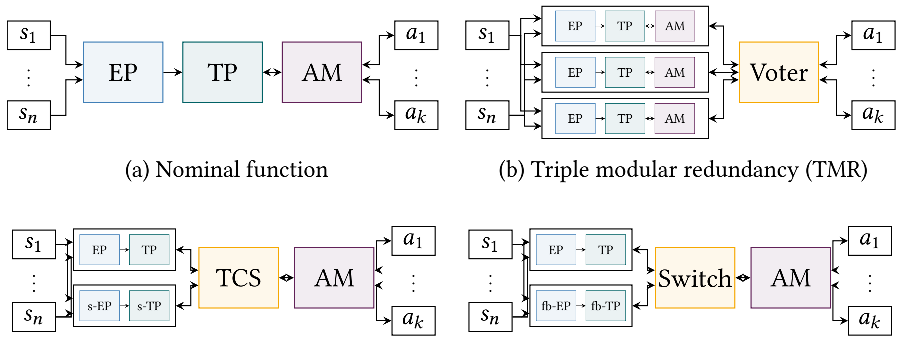
Contributions | The main contribution of this case study is two-fold: we report on the usage of dynamic fault trees for safety analysis in a potential automotive setting. While standard fault tree analysis is part of the ISO 26262, the usage of DFTs in this field is new. We show how the additional features offered by DFTs help to create faithful models of the considered scenarios. These models are then used to analyse the given scenarios. To increase the applicability of DFTs as a method for probabilistic safety assessment in an industrial setting, we give concrete building blocks to work with, e.g. redundancy and faults covered by fallible safety mechanisms.

A clear benefit of the usage of DFTs is that all these methods are integrated in existing off-the-shelf analysis tools which provide sound error bounds. The usage of DFTs reduces the amount of domain-specific knowledge in the analysis, and thus, supports a more model-oriented approach. In this case study, we use the model-oriented approach to investigate the effect of different hardware partitioning on a range of metrics. The generated DFTs are as far as we know the largest real-life ones in the literature—larger trees have only been artificially created for scalability purposes [JGKR+17].

> **Remark 8.1** This work has been carried out in cooperation with BMW AG. The proposed concepts and architectures are exemplary for real-life systems. No implication on actual safety concepts or E/E architectures implemented by BMW AG can be derived from these examples. The same remark applies on any quantity (failure rates, obtained metrics, etc.) presented in this chapter.

8.2 Vehicle guidance

The most challenging safety topic in the automotive industry is currently the driving automation, where the driving responsibility is moving partly or even entirely from the



(a) Nominal function

(b) Triple modular redundancy (TMR)

(c) Nominal path and safety path

(d) Main path and fallback path

Figure 8.2: Different functional block diagrams for vehicle guidance

driver to the embedded vehicle intelligence. Rising liability questions make it crucial to develop functional safety concepts adequately to the intended automation level and to provide evidence regarding the availability and the reliability of these concepts.

8.2.1 Scenario

As a real-life case study from the automotive domain, we consider the *functional block diagram (FBD)* in Figure 8.2a representing the skeletal structure of automated driving. Data collected from different sensors s_i—cameras, radars, ultrasonic, etc.—are synthesised and fused to generate a model of the current driving situation in the Environment Perception (EP) functional block. This model is used by the Trajectory Planning (TP) functional block to build a driving path with respect to the current driving situation and the intended trip. The Actuator Management (AM) functional block ensures the control of the different actuators a_i—powertrain, brakes, etc.—following the calculated driving path. Thus, the blocks in the FBD fulfil tasks: the tasks are realised by (potentially redundant) functional blocks, connected by lines to depict dataflow. We like to stress that these diagrams are not *reliability block diagrams* [TB17] in which the system is operational as long as a path through operational blocks exist.

8.2.2 Modelling safety concepts

8.2.2.1 Technical safety concepts

Based on the criticality of the vehicle guidance function—especially when the driver is out-of-the-loop—the highest level ASIL D applies to the safety goal of following a safe trajectory. According to the automation level, the vehicle guidance function must be designed as *fail-operational*, i.e. the system should safely continue to operate for a certain time after a failure of one of its components. Different design patterns have been developed and implemented in safety-critical systems with fail-operational behaviour and high safety levels, cf. e.g. [ASK09]. The variety of possibilities is illustrated by the following three concepts:

SC1 - Triple Modular Redundancy (TMR) (Figure 8.2b): The nominal function for vehicle guidance is replicated into three paths each fulfilling ASIL B. A Voter, fulfilling ASIL D, ensures that any single incorrect path is eliminated.

SC2 - Nominal path and safety path (Figure 8.2c): Consists of two different paths, a nominal path (n-Path) and a safety path (s-Path) in hot-standby mode. The n-Path provides a full extent trajectory—including comfort functions not necessary for safe operation—with ASIL QM and the s-Path a reduced extent trajectory but with highest safety integrity level ASIL D. The reduced extent safety trajectory is generated from a reduced s-EP (safety Environment Perception) and reduced s-TP (safety Trajectory Planning). The Trajectory Checking and Selection (TCS) verifies whether the trajectory calculated by the n-Path is within the safe range calculated by the s-Path or not. In the case of failure, the s-Path takes over the control and the safe trajectory with reduced extent is followed by the AM. In this case, the system is considered to be *degraded*.

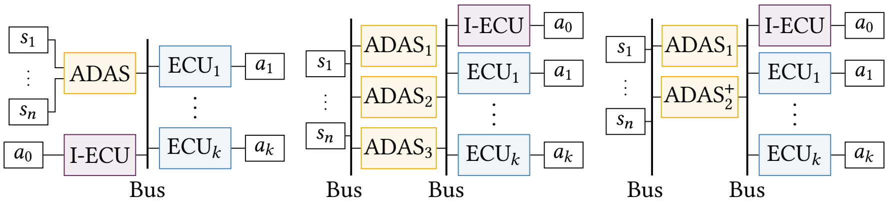
SC3 - Main path and fallback path (Figure 8.2d): Similar to SC2 although the main path (m-Path) is now developed according to ASIL D in order to detect its own hardware failures and signalise them to the Switch. The Switch then commutates the control of the AM to a fallback path (fb-Path), operated in cold-standby, with ASIL D. Upon activation of the fb-Path, the system is considered to be *degraded*.

8.2.2.2 Partitioning on E/E architecture

The next design step consists of extending the nominal E/E architecture for vehicle guidance and partitioning the blocks of every safety concept on its elements. The nominal E/E architecture is represented in Figure 8.3a. The vehicle guidance function is implemented on an ADAS-platform (Advanced Driver Assistance System) which is connected to all sensors. A number of dedicated ECUs (Electronic Control Unit) control the actuators. On an I-ECU (Integration ECU), additional, non-dedicated actuation functions can be implemented. Naturally, implementing all blocks from the safety concepts on the ADAS in Architecture A defeats the purpose of the redundant paths.

E/E architectures for safety concepts | Figure 8.3 gives further illustrative examples for E/E architectures for the different safety concepts.

For **SC1**, Architecture B (Figure 8.3b) allows an implementation of the three redundant paths on separate ADAS-cores. The Voter is then implemented on the I-ECU.



(a) E/E architecture A (b) E/E architecture B (c) E/E architecture C

Figure 8.3: Different E/E architectures

For **SC2**, the following two implementations both yield ASIL D for the safety path, each with TCS and AM on the I-ECU:

1. Executing the nominal path on one ADAS and redundant execution of the s-Path on two ADAS-cores in lock-step mode, using Architecture B.
2. Encoded execution [GKSF16] of the s-Path on a single ADAS+-core in Architecture C (Figure 8.3c), where the + refers to the additional hardware resources to run an encoded s-Path.

For **SC3**, the E/E architecture could be realised on Architecture C, where the m-Path is implemented on ADAS$_1$ and the fb-Path on ADAS$_2^+$.

Alternatives are considered in our experiments in Section 8.5.

8.2.2.3 Hardware platforms and faults

We assume that all hardware platforms can completely recover from *transient faults* (e.g. by restarting the affected path), so that only transient faults directly leading to a system failure are of importance. Transient faults quickly vanish. Thus, the probability of an additional transient fault occurring is small. We assume that this probability is negligible and during a transient fault no other faults occur, cf. [ISO26262]. It can be understood as modelling only single transient faults.

8.2.3 Measures

The safety goal for the considered systems is to avoid wrong vehicle guidance, i.e. following unsafe trajectories. As the system is designed to be *fail-operational*, the system should be able to maintain its core functionality for a certain time, e.g. 10 seconds—even in the presence of faults. The safety goal is violated, if e.g. two out of three TMR paths or both the n-Path and the s-Path fail. The goal is also violated if e.g. a failure of the n-Path is not detected. The safety goal is classified as ASIL D. We stress that safe faults do not need to be considered.

Several measures allow insights in the safety-performance of the different safety concepts: *System reliability* refers to the probability that the system safely operates during the considered operational lifetime. To obtain the average failure-probability per hour, the *unreliability* of the system is scaled with the lifetime. Besides the reliability, the *mean-time-to-failure (MTTF)* is a standard measure of interest. We also consider *degraded states* in which some faults already occurred in the system. If the system is in a degraded state, it still safely operates but provides reduced functionality. The following measures focusing on degraded states reflect insights also relevant for customer satisfaction:

1. the probability that the system provides the full functionality at time t,
2. the fraction of system failures occurring without being in a degraded state before,
3. the expected time to failure upon entering a degraded state,
4. the criticality of a degraded state in terms of the probability that the system fails within e.g. a typical drive cycle of one hour (cf. [ISO26262, 5:9.4]) while being degraded already, and
5. the effect on the overall system reliability when imposing limits on the time a system remains operational in a degraded state.

It is important to consider the robustness or *sensitivity* of all measures w.r.t. changes in the failure rates. Furthermore, it is beneficial for a proper analysis to consider the system under (hypothesised) combinations of events.

8.3 Creating the dynamic fault trees

This section describes in detail our approach as depicted in Figure 8.1 on page 268. In particular, we discuss how to systematically create a DFT consisting of three layers. The top-level event is assumed to represent a safety-critical failure of the vehicle-guidance system. From top to bottom, the layers are:

1. the *system layer* describes how the top-level event occurs due to failing function blocks (e.g. EP or TP). Thus, the layer's basic events describe the failure of a function block. ·system layer
2. the *block layer* describes how a function block can fail, taking into account the possibility of failing computation, or wrong inputs. Thus, the layer's basic events describe failures of the E/E-architecture. In particular, either failures of the hardware platform on which the function block is executed, or the failure of buses that realise the communication between function blocks. ·block layer
3. the *hardware layer* describes how the hardware platforms can fail, based on failures of hardware components, which are considered as basic events. ·hardware layer

The system- and block layer depend on the functional architecture of the system, in particular on the structure and information captured in a (given) functional block diagram. The fault trees for the hardware are typically constructed by the manufacturers.

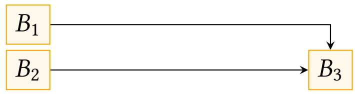


Figure 8.4: Toy-example function block diagram

The connections between the block layer and the hardware layer are inferred from the E/E-architecture and the hardware partitioning.

8.3.1 From function block diagram to the system- and block layer

We discuss the creation of the system- and block layer of the fault tree, based on a functional architecture and a function block diagram for this architecture.

block diagram

Definition 8.1 (Block diagram) A *block diagram* $\mathcal{D} = (\mathsf{Blc}, \rhd)$ is a finite directed graph. The vertices Blc are a set of blocks, the edges $\rhd$ are called channels.

Given a channel $(B, B') \in \rhd$, we call B the source and B' the target. The input and output channels of block B are denoted by $\mathsf{in}_B = \{e \in \rhd \mid e \in \mathsf{Blc} \times \{B\}\}$ and $\mathsf{out}_B = \{e \in \rhd \mid e \in \{B\} \times \mathsf{Blc}\}$, respectively. The block diagram may contain cycles, cf. Figure 8.6b.

Within the scope of this case study, we advocate the structured manual creation of the system- and block layer. A further discussion is given in Section 8.6.2. Below, we discuss the creation of both layers.

Example 8.1 (Function block diagram) The function block diagram in Figure 8.4 is formally given by $\mathsf{Blc} = \{B_1, B_2, B_3\}$, and $\rhd = \{(B_1, B_3), (B_2, B_3)\}$.

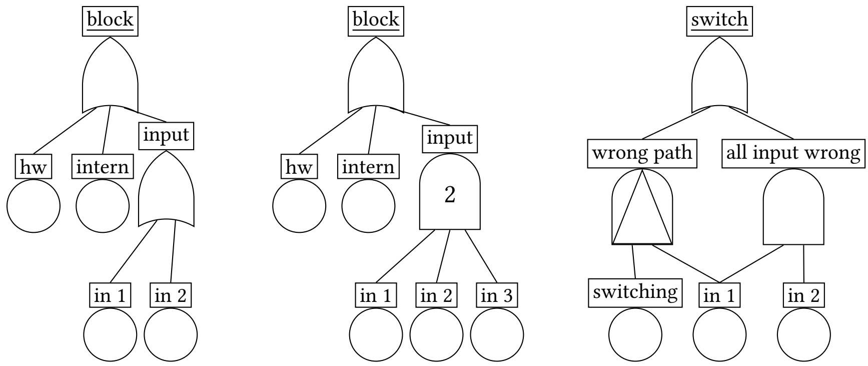
Remark 8.2 Typically, fault trees are created top-down. We deviate from this scheme to ease the presentation, and show the potential for automation.

8.3.1.1 Creation of the block layer

block fault tree

For the systematic construction of the block layer, we assume that the possible causes of a block to fail are described by an appropriate *block fault tree*. The causes include faulty input channels.

Creating fault trees for each block | In a first step, we create a fault tree $\mathcal{F}^B$ for each block $B \in \mathsf{Blc}$ in the block diagram. To reflect the interaction within both the design pattern and the influence from the assigned hardware later on, we annotate each block fault tree $\mathcal{F}^B$ with relevant information:



(a) Block FT - General (b) Block FT - Voter (c) Block FT - Switch

Figure 8.5: Block fault trees

- For each input channel of B, we define a BE for faulty input.
- For each output channel c of B, we define an element in $\mathcal{F}^B$ which causes B's output on c to be faulty. Later, we propagate the faulty output to the block fault trees for the target block along the channel c.
- We define a BE for the failure of the hardware platform that executes B.

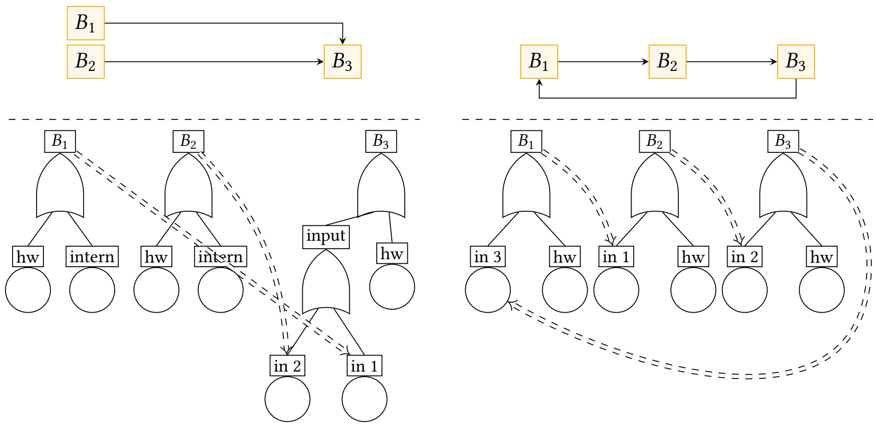
Definition 8.2 (Block fault tree) A *block fault tree* for block B is a tuple $(\mathcal{F}^B, I^B, O^B, Y^B)$, where

block fault tree

- $\mathcal{F}^B$ is a fault tree,
- $I^B : \mathsf{in}_B \longrightarrow \mathsf{BE}_{\mathcal{F}^B}$ denotes input faults,
- $O^B : \mathsf{out}_B \longrightarrow \mathcal{F}^B$ denotes output failures, and
- $Y^B \in \mathsf{BE}_{\mathcal{F}^B}$ denotes the hardware fault BE.

A fault tree for a block B with two input channels (c_1, c_2) and one output channel (c_3) is given in Figure 8.5a. The block fails if either the hardware fails, due to some internal fault, or due to faulty input. Typically, the hardware failure (formally, $Y^B = \mathsf{hw}$) is connected to the hardware fault tree, see Section 8.3.2.1. The internal fault can be used to support sub-blocks or for additional fallible events. Faulty input means the failure of any of the inputs (formally, $I^B(c_1) = \mathsf{in\ 1}$, $I^B(c_2) = \mathsf{in\ 2}$). The output is faulty, if the block fails (formally, $O^B(c_3) = \mathsf{block}$). Adaptions of the standard scheme are possible:

- For the voter block, as in triple modular redundancy SC1: We assume that the voter fails if two out of three inputs are faulty. Thus, we obtain a block fault tree as depicted in Figure 8.5b, with three inputs. The block fails if two out of three inputs fail, as reflected by a VOT_2-gate.
- For the switch as in SC3: The switching mechanism may fail. This only leads to



(a) Toy block diagram and connected block FT (b) Feedback loop and connected block FT

Figure 8.6: Connecting block fault trees

a failure when the path has to be switched though. This behaviour is reflected by the DFT in Figure 8.5c: the switch fails if it either uses the wrong path or if all input is wrong. The path is wrong if the switching mechanism fails before the primary input in 1 fails, i.e. it can no longer switch to an operational path, as reflected by a PAND-gate.

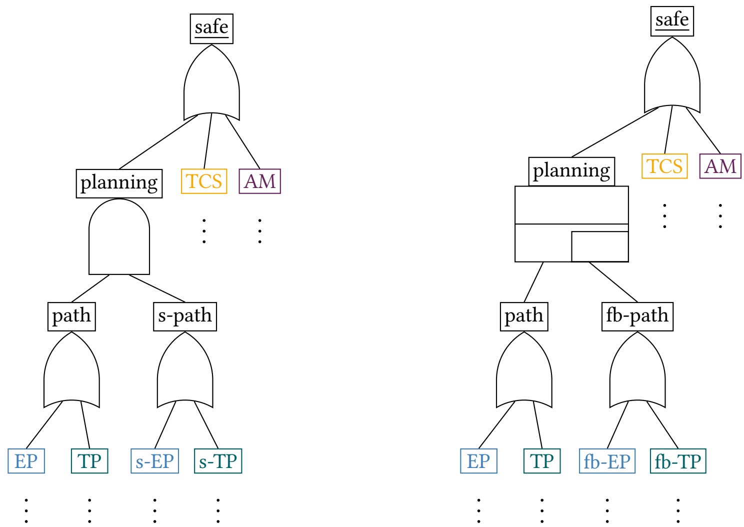
Block fault trees can easily be extended to several hardware failures, to input/output faults of several types, etc.

·connected block fault tree

Connecting block fault trees | The goal of this step is to connect the inputs and outputs of the block fault trees as specified by the relation $\rhd$. The *connected block fault tree* consists of the disjoint union of all block fault trees. Additionally, for each channel $c = (B, B')$ in the block diagram, we connect the output failure $O^B(c)$ with the input fault $I^{B'}(c)$. The connection is realised by means of an FDEP with trigger $O^B(c)$ and dependent event $I^{B'}(c)$. As we only consider the block layer, there is no dedicated TLE.

Example 8.2 (Connected block fault tree) We consider the toy example of Example 8.1. We illustrate the corresponding connected block fault tree in Figure 8.6a. For each block, we use a standard block as in Figure 8.5a. We assume a faulty input to the incoming channels of B_3 if the respective source block fails, e.g. input in 2 is failed if block B_2 failed.

A major benefit of FDEPs is that they allow to faithfully model feedback loops,



(a) System layer FT for SC2 (b) System layer FT for SC3

Figure 8.7: Task-based construction of the system layer

cf. Section 3.7. We illustrate the support for feedback loops in Figure 8.6b: For three blocks as shown on top, the three DFTs are connected via FDEPs. If, e.g. B_1 fails, the failure is propagated to the input of B_2, etc.

8.3.1.2 Creation of the system layer

The goal of this step is to express the occurrence of a safety-critical failure as a fault tree over the failure of the blocks. Towards this goal, we process top-down. A task-based partitioning of the DFT is helpful. The fault tree is assumed to fail if any of the tasks can no longer be executed. We model this with an OR-gate as top-level event. The tasks fail if no block can realise the task anymore, modelled with AND-gates. We do not need to consider error propagation through the blocks at this point, as this is already handled by the connections—via FDEPs—in the connected blocks fault tree.

We depict the system layer fault tree for SC2 in Figure 8.7a: the system fails, if either the planning, the selection, or the actuator management fails. Planning fails if both the nominal and the safety path fail.

To model cold/warm redundancy, we replace AND by the more general SPARE-gate. We illustrate this for SC3 in Figure 8.7b. Recall that the fb-Path is in cold standby. EP and TP in the fallback-path only operate as soon as they are required, i.e. as soon as the

8

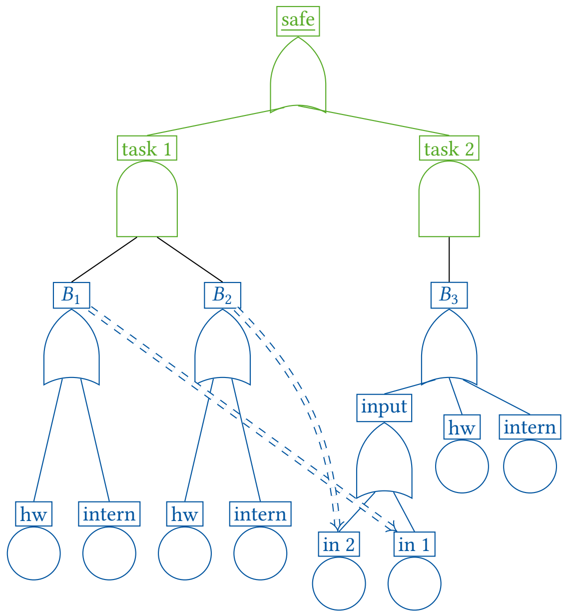


Figure 8.8: Putting the system layer (green) and the block layer (blue) together

nominal path fails. This behaviour is captured by using a SPARE instead of an AND.

> **Example 8.3 (System- and block FT)** We continue Example 8.2. Assume that the blocks B_1 and B_2 execute the same task with B_2 in hot standby—modelled by an AND—and B_3 executes another task. The resulting DFT is depicted in Figure 8.8. Green colour indicates the system layer and blue colour the block layer.

8.3.2 Adding the hardware layer to the system- and block layers

In this section, we discuss how to combine the system- and block layer fault tree with the hardware layer. We assume an E/E-architecture, a hardware assignment and additional input fault trees for all hardware components. Below, we first discuss the additional inputs and then consider how to combine them to a complete fault tree.

8.3.2.1 Fault trees for hardware platforms and buses

We assume that the (D)FTs to model hardware failures are provided by the manufacturers, and we do not make any assumptions about the structure of these fault trees. We

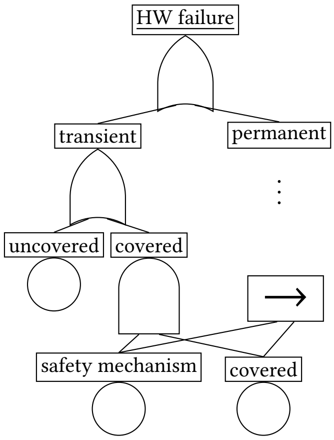


Figure 8.9: Hardware fault tree

briefly illustrate how to integrate coverage and both transient and permanent faults in DFTs, analogous to [ISO26262, 10:B]. We would like to stress the dynamic nature of coverage by fallible safety mechanisms. This is in contrast to e.g. the *coverage* specified in [DVG97], where the safety mechanism for covered faults is always successful.

Consider the DFT depicted in Figure 8.9. A failure results from either a transient or permanent error. Each type has its own corresponding safety mechanism. We consider the transient error here, the permanent error is modelled similarly. A transient error occurs if either the transient fault is not covered by the safety mechanism, or the fault is covered but the safety mechanism has failed before. We model the latter by a SEQ[1]. It can neither be modelled faithfully by an SFT nor a PAND—though for other analysis purposes using a PAND might be more adequate. A PAND would model that a covered error is never propagated if the covered fault occurs before.

8.3.2.2 Hardware assignment and E/E architecture

We briefly discuss the assumptions about the hardware assignment and the E/E architecture. We formalise the concepts as follows.

> **Definition 8.3 (E/E-architecture)** An E/E-architecture is a tuple (H, Bus) of a set H of hardware platforms, and a set Bus of buses. Each bus is a transitive relation over hardware platforms, i.e. $\mathrm{Bus} \subseteq 2^{H \times H}$.

Often, buses are equivalence relations, i.e. they connect a set of platforms to each

[1] SEQs also appear in the ISO 26262 [ISO26262, 10-B.3], where they are indicated by the boxed L.

other, as e.g. in FlexRay [ISO17458] or CAN [ISO11898]. We assume that all hardware platforms $p \in H$ contain an internal bus $\mathsf{internal}_p = \{(p,p)\} \in \mathsf{Bus}$. Consequently, function blocks on the same hardware platform can always communicate.

Example 8.4 (E/E architecture) We formalise the E/E-architecture B from Figure 8.3b on page 272. The hardware platforms H are:

$$\{s_1, \ldots, s_n, \mathrm{ADAS}_1, \mathrm{ADAS}_2, \mathrm{ADAS}_3, \text{I-ECU}, \mathrm{ECU}_1, \ldots, \mathrm{ECU}_k, a_0, \ldots, a_k\},$$

and Bus is given as

$$\{\text{ECU-actuator}_0, \ldots, \text{ECU-actuator}_k, \text{CAN-BUS}\} \cup \left\{\mathsf{internal}_p \mid p \in H\right\},$$

with
- $\text{ECU-actuator}_0 = \{(\text{I-ECU}, a_0), (a_0, \text{I-ECU})\}$,
- $\text{ECU-actuator}_i = \{(\mathrm{ECU}_i, a_i), (a_i, \mathrm{ECU}_i)\}$, and
- $\text{CAN-BUS} = (H \setminus \{a_0, \ldots, a_k\}) \times (H \setminus \{a_0, \ldots, a_k\})$.

hardware assignment

Definition 8.4 (Hardware assignment) Given a block diagram $\mathcal{D} = (\mathsf{Blc}, \rhd)$ and E/E-architecture (H, Bus), a *hardware assignment* is a function $h : \mathcal{D} \to H \cup \mathsf{Bus}$ such that $h(\mathsf{Blc}) \subseteq H$ and $h(\rhd) \subseteq \mathsf{Bus} \cup H$.

·consistent

More precisely, hardware assignments map internal channels to hardware platforms, and all other channels to buses. A hardware assignment is *consistent*, if the source and target of each channel are connected by a bus. Thus, the assignment is consistent if

$$(B, B') \in \rhd \implies \exists C \in \mathsf{Bus} \text{ with } (h(B), h(B')) \in C.$$

A hardware assignment trivially supports mapping several function blocks to the same hardware. In this case, we assume that a failure of a function block does not affect other function blocks on the same hardware. If necessary, dependent failures of function blocks can be modelled explicitly in the DFT using FDEPs. We require that any function block runs on at most one hardware platform. If the same function should be implemented on several hardware platforms, we require an explicit duplication in the function block diagram.

Remark 8.3 We do not consider whether a hardware assignment is feasible, i.e. whether the hardware platforms provide the (computational) resources to realise the functions.

Block-based assignments | Typically, we are only given an assignment from blocks to hardware platforms; the channel assignment then follows from the E/E architecture.

That is, we assume that any channel between two blocks assigned to the same hardware platform is realised by the internal bus. For channels that connect blocks which are assigned to different platforms, we select the unique bus (e.g. the CAN-bus) that connects these two platforms. If there is no unique bus, then manual intervention is required to specify the proper bus.

Example 8.5 (Hardware assignment) Consider the block diagram of Example 8.2 on page 276, and the E/E-architecture as in Example 8.4. The mapping of the blocks $h(B_1) = \text{ECU}_1$, $h(B_2) = h(B_3) = \text{ADAS}_1$ induces a unique mapping of the channels according to the scheme described above:

$$h((B_1, B_3)) = \text{CAN-BUS} \quad \text{and} \quad h((B_2, B_3)) = \mathsf{internal}_{\text{ADAS}_1}.$$

8.3.2.3 Constructing a complete fault tree

To obtain a complete DFT of the vehicle guidance system, we take the disjoint union of the DFT $\mathcal{F}$ for the system and block layer, and the hardware DFTs $\{\mathcal{F}_y \mid y \in H \cup \mathsf{Bus}\}$. The DFT $\mathcal{F}$ contains annotations Y^B for each block B. For each $B \in \mathsf{Blc}$, an FDEP from the top-level event of $\mathcal{F}_{h(B)}$ to the dependent event Y^B is added. The FDEP ensures that a failure of the hardware platform causes the function blocks executed on that hardware to fail. To ensure that the hardware FTs are correctly activated if the corresponding function blocks are activated, we replace selected FDEPs with OR-gates, cf. Section 4.1.2. Furthermore, for each channel $c = (B, B') \in \triangleright$, an FDEP from the top-level element of $\mathcal{F}_{h(c)}$ to the dependent event $I^{B'}(c)$ is added. The FDEP ensures that if the bus fails, the input of the target block fails as well.

Example 8.6 (Complete FT) We connect the fault tree for the system and block layer from Example 8.3 on page 278 with hardware FTs, cf. Figure 8.9. The connection is based on the hardware assignment in Example 8.5. The relevant fragment of the complete DFT is depicted in Figure 8.10. Green colour indicates the system layer, blue colour the block FT and red colour the hardware FT.

8.4 Analysing the dynamic fault trees

Our aim is to analyse the complete fault trees constructed as in Section 8.3 with respect to the eight measures described in Section 8.2.3. Our analysis uses Storm-dft with all its optimisations, cf. Chapter 5. The full tool-chain follows the approach previously depicted in Figure 5.6 on page 197. First, the DFT is simplified via rewriting, cf. Section 5.4.5, and converted into a CTMC. Second, the safety measure is translated

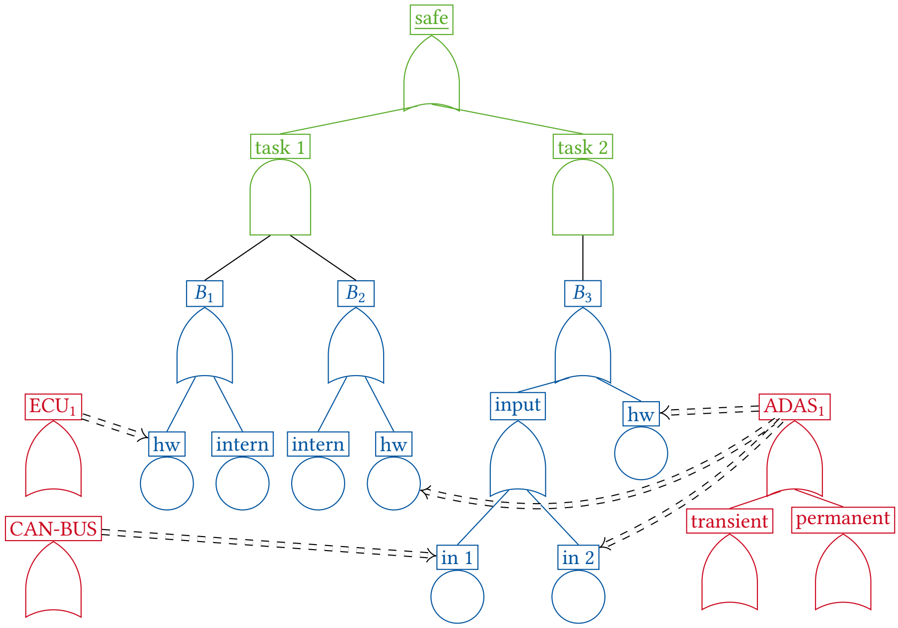


Figure 8.10: Fragment of a complete fault tree for a toy example

into a model-checking query as detailed in Section 8.4.1. Finally, the result for the query on the CTMC is computed based on traditional probabilistic model checking extended with some optimisations as detailed in Section 8.4.2.

Non-determinism | Note that when using the default translation, the resulting model is a *Markov automaton*. The non-determinism stems from the *cyclic dependencies* in the DFT. However, in our setting, the order of FDEP failures does not influence the obtained analysis result. Therefore, we fix a failure order for dependent events and restrict ourselves to CTMCs.

Transient faults | *Transient faults* are considered during the state-space generation. Recall from Section 8.2.2.3, that if a transient fault occurs, either the system fails directly, or the fault disappears and the system returns to its previous state. During the state-space generation, transient faults are considered in each state similar to regular BE faults. However, the corresponding transition is only added to the CTMC if the TLE has failed in the target state of the transition. Otherwise, the failure of this specific BE is ignored—similar to how *invalid states* for SEQs are handled, cf. Section 5.2.2.2.

Table 8.1: Model-checking queries

	Measure	Model-checking queries
System	Reliability	$1 - \mathbf{P}\left(\Diamond^{\leq t}\ \mathit{Failed}\right)$
	AFH	$\frac{1}{\text{lifetime}} \cdot \mathbf{P}\left(\Diamond^{\leq \text{lifetime}}\ \mathit{Failed}\right)$
	MTTF	$\mathbf{ET}\left(\Diamond\ \mathit{Failed}\right)$
Degradation	FFA	$1 - \mathbf{P}\left(\Diamond^{\leq t}\ (\mathit{Failed} \vee \mathit{Degraded})\right)$
	FWD	$\mathbf{P}\left(\neg \mathit{Degraded}\, \mathrm{U}^{\leq t}\ (\neg \mathit{Degraded} \wedge \mathit{Failed})\right)$
	MTDF	$\sum_{s \in \mathit{Degraded}} \left(\mathbf{P}\left(\neg \mathit{Degraded}\, \mathrm{U}\ s\right) \cdot \mathbf{ET}_s\left(\Diamond\ \mathit{Failed}\right)\right)$
	MDR	$\text{argmin}_{s \in \mathit{Degraded}} \left(1 - \mathbf{P}_s\left(\Diamond^{\leq t}\ \mathit{Failed}\right)\right)$
	FLOD	$\sum_{s \in \mathit{Degraded}} \left(\mathbf{P}\left(\neg \mathit{Degraded}\, \mathrm{U}^{\leq t}\ s\right) \cdot \mathbf{P}_s\left(\Diamond^{\leq \text{drivecycle}}\ \mathit{Failed}\right)\right)$
	SILFO	$1 - (\text{FWD} + \text{FLOD})$

8.4.1 Converting measures

We convert the safety measures introduced in Section 8.2.3 into model-checking queries composed of standard *CSL* properties (cf. Section 2.3.2). The model-checking queries for each of the eight safety measures is given in Table 8.1.

As before, the label *Failed* denotes states where the top-level event in the DFT has failed. Similarly, *Degraded* denotes states where only reduced functionality is available.

The first three model-checking queries consider the safety-performance of the complete system. For *reliability* and *mean-time-to-failure (MTTF)*, we use the standard properties already defined in Section 5.3.1. The average failure-probability per hour (AFH) is obtained similar to the reliability by using the lifetime of the system as time bound and afterwards normalising by the lifetime.

The second set of queries describes the influence of degradation in the system.

1. Full Function Availability (FFA) describes the time-bounded probability that the system provides full functionality, i.e. it is neither failed nor degraded. It is described as the complement of the time-bounded reachability of a failed or degraded state.
2. Failure Without Degradation (FWD) describes the time-bounded probability that the system fails without being degraded first. It is the time-bounded probability of reaching a failed state without reaching a degraded state.
3. Mean Time from Degradation to Failure (MTDF) describes the expected time from the moment of degradation to system failure. It is obtained by taking the expected time of failure for each degraded state and scaling it with the probability

to reach this state while not being degraded before.

4. Minimal Degraded Reliability (MDR) describes the criticality of degraded states by giving the worst-case failure probability when using the system in a degraded state. For all degraded states, the time-bounded reachability of a TLE failure is computed. The MDR is the minimum over the complement of this result for all degraded states.
5. System Integrity under Limited Fail-Operation (SILFO) considers the system-wide impact of limiting the degraded operation time. SILFO is split into two parts considering failures without degradation (FWD) and failures with degradation (FLOD). Failure under Limited Operation in Degradation (FLOD) describes the probability of failure when imposing a time limit for using a degraded system. For all degraded states the time-bounded reachability probability of a failed state is computed within the restricted time-bound given by a drive cycle. This value is scaled by the time-bounded probability of reaching a degraded state without degradation before.

For *sensitivity* analysis, we perform several model-checking queries on CTMCs of DFTs with different failure rates.

8.4.2 Analysis via model checking

The first five measures are computed with a single model-checking query each. Furthermore, it is possible to compute (time-bounded) reachability probabilities starting in a state s for all states $s \in S$ simultaneously. Thus, MDR can be obtained by computing $\mathbf{P}_s(\Diamond^{\leq t}\, \mathit{Failed})$ for all degraded states s in a single query and afterwards computing the minimum over the complement of all results.

However, the computation of MTDF and SILFO requires the computation of (time-bounded) until probabilities reaching a degraded state s for all degraded states $s \in S$ independently. A naïve implementation would require a model-checking query for each degraded state, increasing the computation time drastically. Using similar ideas as in [KKNP01], we implemented an improved algorithm computing (time-bounded) until probabilities for all states simultaneously. The improved algorithm works as follows.

Improved algorithm for multiple target states | The standard approach for until probabilities is a 'backwards' computation of the probabilities starting from the target states we want to reach. The result is a vector of the (time-bounded) probabilities to reach a target state from each state. The idea here now is to perform a 'forwards' computation of the probabilities starting from the initial state. The result is a vector of (time-bounded) probabilities to reach each target state from the initial state. Note that this algorithm applies a transformation which makes all target states absorbing, i.e. they only have self-loops. That means in particular that target states reachable via other target states will be reached with lower—or zero—probability now and the obtained probability is not correct for these target states. In our scenario however, we are only interested in the first occurrence of a degraded state and therefore can make each degraded state absorbing without affecting the results.

The improved algorithm performs a model-checking query only once for all degraded states at the same time. Thus, the computation of MTDF can be reduced to performing one model-checking query for $\mathbf{P}(\neg \mathit{Degraded} \,\mathsf{U}\, s)$, one query for $\mathbf{ET}_s(\lozenge\, \mathit{Failed})$ and combining the results afterwards. The computation of SILFO can be performed similarly by combining the results of the three model-checking queries—for FWD, for $\mathbf{P}\left(\neg \mathit{Degraded} \,\mathsf{U}^{\leq t}\, s\right)$ and for $\mathbf{P}_s\left(\lozenge^{\leq \text{drivecycle}}\, \mathit{Failed}\right)$.

Evidence | To flexibly support the analysis of degrades states, we use the concept of *evidence*. Evidence is given as a set of BEs considered as already failed and all possible ftraces of these BEs are examined. Following the traces from the initial state yields a set S' of states describing the system status based on the given evidence. Computing a model-checking query using evidence gives results starting in each state $s \in S'$. The same behaviour can also be directly modelled in the DFT by introducing a $\mathsf{CONST}_\top$ and adding FDEPs from the $\mathsf{CONST}_\top$ to each BE in the evidence. However, this approach requires modifying the DFT for each new set of evidence. When the complete state space has been built before, it is beneficial to reuse it for analysing the degraded system. evidence

8.5 Experiments

We show the applicability of the proposed methodology on systems and concepts similar to those from Section 8.2.2. The generated (anonymised) DFTs are available online[2] and also part of the FFORT benchmark collection [RBNS+19] under the name Vehicle Guidance System (VGS).

8.5.1 Set-up

All experiments were executed on a 2.9 GHz Intel Core i5 with 8 GB of memory and a time limit of 1 h. We consider the three safety concepts SC1, SC2 and SC3. For each safety concept, we construct a system- and block layer fault tree as described in Section 8.3.1. All scenarios include four sensors, of which at least two are required for safe operation, and four actuators, which are all required for safe operation.

8.5.1.1 Different partitioning schemes

We discuss several partitioning schemes for the safety concepts, which we refer to as scenarios. Each scenario is defined by the safety concept, the used architecture with

[2] https://www.stormchecker.org/publications/dfts-for-vehicle-guidance-systems

Table 8.2: Scenarios

Scenario	Safety concept	Architecture	Adaptions	Sensors	Actuators
I	SC1	B	—	2/4	4/4
II	SC2	B	—	2/4	4/4
III	SC2	C	ADAS+	2/4	4/4
IV	SC3	C	—	2/4	4/4
V	SC2	A	—	2/4	4/4
VI	SC2	B	removed I-ECU	2/4	4/4
VII	SC2	B	5 ADAS, 2 BUS	2/8	7/7
VIII	SC2	B	8 ADAS, 2 BUS	2/8	7/7

possible adaptions, the hardware assignment, and the fraction of sensors and actuators which have to be operational. An overview of the considered scenarios is given in Table 8.2. We consider E/E-architectures A, B and C from Figure 8.3 on page 272, and vary the architectures, by e.g. considering a redundant bus or introducing more ADAS platforms. Additionally, we consider partitions which do not use the I-ECU, and instead assign functions to the ADAS_2. Furthermore, we scale the number of sensors and actuators and the required number of sensors for safe operation.

8.5.1.2 Failure rates

For presentation purposes we assume the following failure rates, which do not necessarily reflect reality and especially do not reflect any system from BMW AG. We assume that function blocks, e.g. EP, are free of systematic (internal) faults. Sensors, actuators, and (I-)ECUs have failure rates of 10^{-7} ¹/h. In the ADAS hardware platforms transient faults occur with rate 10^{-4} ¹/h and permanent faults occur with rate 10^{-5} ¹/h. All faults can be detected by a safety mechanism which itself fails with rate 10^{-5} ¹/h. The safety mechanism for ASIL D covers 99 % of the faults, the safety mechanism for ASIL B covers 90 %, and the one for ASIL QM covers 60 % of the faults. For the ADAS+ platform, the failure rates increase by a factor 10 and the safety mechanism covers 99.9 % of all faults.

8.5.1.3 Tool-support

The complete workflow consisting of both the generation of the DFTs as well as their analysis, are supported by a Python tool-chain. Given a safety concept as a function block diagram and the FT for each block, the tool-chain first automatically generates a FT where dependencies are inserted according to the data flow in the safety concept. Given an E/E architecture with a partitioning and hardware FTs, and the system- and

Table 8.3: Model characteristics

Scenario	**DFT**			**CTMC**		
	#BE	#Dynamic	#Elements	#States	#Transitions	% Degraded
I	76	25	233	5377	42 753	–
II	70	23	211	5953	50 049	19 %
III	57	19	168	1153	7681	17 %
IV	57	21	170	385	1985	12 %
V	58	19	185	193	897	0 %
VI	65	21	199	1201	8241	20 %
VII	96	30	266	109 369	1 148 785	19 %
VIII	114	36	305	5 179 105	84 454 945	11 %

block layer FT generated before, the complete FT is automatically constructed. This generation of the complete FT is performed in milliseconds. Finally, the analysis of the complete FT is performed fully automatically as well.

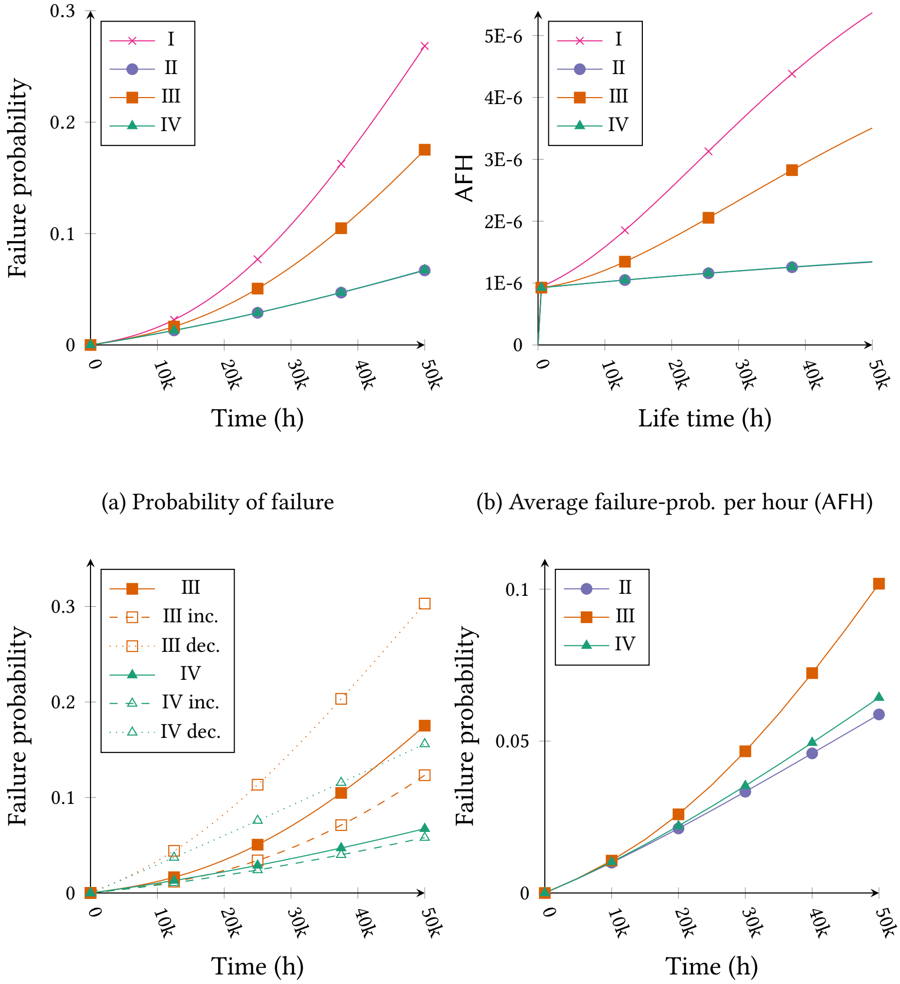
We describe the failure rates in the hardware FTs symbolically, i.e. as *parameters*. Thus, changes in coverage or failure rates require only a different instantiation of the parameters, instead of reconstructing the FT.

8.5.2 Evaluation

In the following we consider the scenarios of Table 8.2. The characteristics of the corresponding DFTs and CTMCs can be found in Table 8.3. The first column identifies the scenario. The following three columns give the number of BEs, dynamic gates and the total number of elements in the DFT. The last three columns describe the CTMC obtained after generating the state space and applying reduction techniques. The columns contain the number of states and transitions, and the percentage of degraded states in the CTMC.

The analysis results for the measures from Section 8.4.1 are given in Table 8.4. Notice that SC1 does not contain degraded states and in scenario V the system fails before reaching a degradation. We use a lifetime of 10 000 h and a drive cycle of 1 h [ISO26262, 5:9.4]. The times (in seconds) for generating the CTMC from the DFT and computing each measure on the CTMC are given in Table 8.5.

Figure 8.11 illustrates the obtained measures for a variety of concepts and architectures. In Figure 8.11a, the failure probability, i.e. the unreliability, over a lifetime of 50 000 h is given for the scenarios I-IV. Figure 8.11b depicts the AFH. Figure 8.11c compares the sensitivity of the failure probability for scenarios III and IV. For the sensitivity analysis we change the ASIL levels of the hardware FTs, i.e. change the



(a) Probability of failure (b) Average failure-prob. per hour (AFH)

(c) Sensitivity analysis (d) SILFO

Figure 8.11: Analysis results

Table 8.4: Obtained measures for different scenarios with operational lifetime of 10 000 h and drive cycle of 1 h (x^c indicates complement $1-x$)

Scen.	System			Degradation					
	rel.^{c}	AFH	MTTF	FFA^c	FWD	MTDF	MDR	FLOD	SILFO^c
I	1.6E−6	1.6E−6	8.6E4	—	—	—	—	—	—
II	1.0E−2	1.0E−6	3.4E5	5.2E−2	1.0E−2	2.3E5	2.9E−1	4.3E−8	1.0E−2
III	1.2E−2	1.2E−6	1.1E5	5.2E−2	1.1E−2	2.1E4	7.4E−1	2.7E−7	1.1E−2
IV	1.0E−2	1.0E−6	3.1E5	1.6E−2	1.0E−2	1.6E5	2.1E−1	6.5E−9	1.0E−2
V	6.0E−2	6.0E−6	6.9E4	6.0E−2	6.0E−2	0	0	0	6.0E−2
VI	1.1E−2	1.1E−6	3.4E5	5.3E−2	1.1E−2	2.3E5	2.1E−1	4.7E−8	1.1E−2
VII	1.7E−2	1.7E−6	2.8E5	5.8E−2	1.7E−2	1.7E5	3.7E−1	7.2E−8	1.7E−2
VIII	1.7E−2	1.7E−6	2.7E5	9.8E−2	1.6E−2	2.0E5	4.3E−1	1.4E−7	1.6E−2

coverage of the safety mechanism. In both scenarios the straight lines are obtained by the baseline failure rates and coverages for the hardware components as given in Section 8.5.1.2. The dashed (dotted) lines are obtained assuming an increased (decreased) coverage according to an increased (decreased) ASIL level, respectively. The graph in Figure 8.11d displays the SILFO for the safety concepts with degraded states.

Results for the approximation are given in Figure 8.12. Figure 8.12a illustrates the lower and upper bound for the failure probability in scenario VIII w.r.t. the computation time. Figure 8.12b depicts the approximation error over the computation time for the largest scenarios. Computing an approximate reliability allowing a 1 % relative error on scenario VIII requires only 206 410 states and could be computed within 12 s.

8.6 Discussion

8.6.1 Analysis of results

We evaluate the obtained results w.r.t. the assumed failure rates. The evaluation is not intended as a recommendation for a specific partitioning scheme, but they illustrate the possibility to perform design space exploration efficiently. In the following we exemplarily evaluate the results of scenarios I-IV. The variety of measures computed allows some insights in the effect of different safety concepts and the role of degradation.

The AFH and MTTF indicate that system reliability of scenarios II and IV are superior compared to III and I. The differences between II and IV are marginal, and III is better than I. These claims can also be deduced from Figure 8.11a and Figure 8.11b. The similarity between I and III stems from the fact that in both cases the system fails if two paths fail, i.e. two out of three in the TMR of I, or normal and safety path in III.

Table 8.5: Timings (in seconds)

	I	II	III	IV	V	VI	VII	VIII
CTMC generation	0.52	0.51	0.08	0.05	0.02	0.10	12.02	2043.78
reliability^c	0.03	0.03	0.00	0.00	0.00	0.00	1.00	82.84
AFH	0.03	0.04	0.01	0.00	0.00	0.01	0.93	157.94
MTTF	0.01	0.01	0.00	0.00	0.00	0.00	0.18	26.43
FFA^c	—	0.02	0.01	0.00	0.00	0.00	0.54	64.41
FWD	—	0.02	0.00	0.00	0.00	0.00	1.46	61.79
MTDF	—	0.48	0.10	0.02	0.02	0.10	10.78	826.73
MDR	—	0.49	0.09	0.02	0.03	0.09	11.02	829.27
FLOD	—	1.08	0.20	0.05	0.04	0.21	28.49	2945.81
SILFO^c	—	1.10	0.20	0.05	0.04	0.21	29.95	3007.60

It is interesting to see that the encoding in ADAS+ for III only marginally improves the reliability, because the better fault coverage of the encoding is outweighed by the higher load on the hardware. In scenario II, all three paths—one normal and two redundant safety paths—have to fail before the complete system fails. In scenario IV, the fallback path has a reduced hardware load as long as the primary path is still operational, leading to a smaller failure rate for this path.

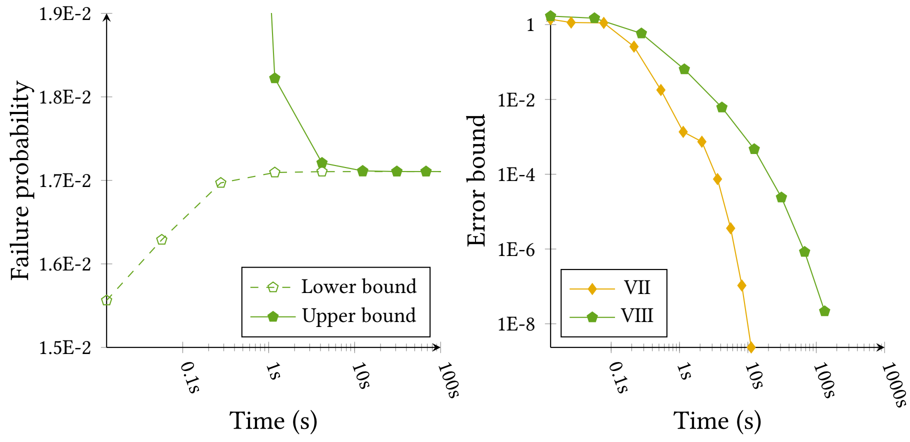
However, from the sensitivity analysis in Figure 8.11c, we can deduce that the lessons are only valid w.r.t. the assumed failure rates. In particular, for scenario IV increasing the safety coverage only has a marginal effect on the failure probability. Thus, the benefit of increasing fault coverage in platforms depends on the chosen architecture.

Scenarios II and IV differ in their failure behaviour of degraded states as seen in Figure 8.11d. When limiting the driving time in the degraded state to one hour, scenario II offers a better reliability than IV whereas in the overall reliability the difference is marginal. The FLOD is orders of magnitude smaller than FWD in all scenarios. Thus, the duration of the drive cycle is insignificant for SILFO.

The results in 8.12 show that we can obtain tight approximation results for the standard measures within seconds, even for the largest scenarios. This is in line with the conclusions of Section 7.2.5.

8.6.2 General methodology

We discuss our methodology based on generating DFTs for function block diagrams and hardware assignments.



(a) Failure probability for VIII

(b) Approximation error

Figure 8.12: Approximation results

Direct translation to CTMCs | A direct translation from the system description to CTMCs is arguably more flexible, and allows to keep any overhead to a minimum. However, even a naïve translation is necessarily complex and error-prone, and the resulting CTMCs are typically too large to be comprehensible. It is hard to give the modeller feedback on the meaning of the constructed CTMC. Fault trees, in contrast, are comparably small, and contain more structure. Additionally, state-space generations have to be implemented with performance in mind, which makes the direct translation likely to be error prone.

Moreover, an additional benefit of using the DFT formalism is due to the presence of tool-support. The state-space generation only had to be slightly adapted—to incorporate *transient faults*—which is significantly easier than the construction of an efficient state-space generation from scratch.

Automation of fault tree generation | Manual creation of the system- and block layer of the fault tree has some advantages, noteworthy are:

- The semantics of the function block diagram do not need to be formalised. In particular, function block diagrams contain implicit assumptions, e.g. assumptions about different failure behaviour of voters, or channels with different meanings. Manual creation can adapt for these subtle differences.
- Constructing the FT is an important step in the development-process of safety-critical systems [ISO26262].

8

Due to the structure, as discussed in Section 8.3, our prototype made several default suggestions, e.g. for block FTs, that greatly reduce the required manual effort.

The generation of the complete fault tree given the system and block layers is fully automated. The automation is essential for the proposed methodology, as this allows a push-button comparison of the various possible variants of hardware assignments.

Using dynamic fault trees | Using DFTs as the underlying model has several advantages. Fault trees are a well-known concept in reliability engineering. Their hierarchical structure allows for a faithful model of the different layers in the considered scenarios. Using DFTs instead of static fault trees provides more expressiveness. For example, most of the proposed measures for degraded states cannot be computed on static fault trees. The proposed fault trees contain several features only present in DFTs, including the gates PAND (for switches), SPARE (for cold redundancy) and SEQ (in safety mechanisms). FDEPs are heavily used, in particular to simplify the representation of feedback loops. Thus, most features of DFTs are present in the generated fault trees and DFTs seem a suitable formalism to assess the failure behaviour.

Analysis methods | Figure 8.11b indicates that the average failure-probability per hour (AFH) varies for different operation life times. This observation justifies the analysis w.r.t. different measures and time horizons. Table 8.3 indicates that reduction techniques successfully alleviate the state-space problem. The generated state space remains small even for hundreds of elements. The size of the state space depends largely on the scenario. Naturally, latent faults increase the state space, but then the effectiveness of the approximation increases as shown in Figure 8.12. Using CTMCs as an underlying model allows to check a wide variety of measures out-of-the-box. Table 8.5 indicates that most of the measures can be computed within seconds even on the largest models. The more complex measures as MTDF and SILFO require a tailored implementation to avoid performing model-checking queries for each degraded state. However, the tailored implementation was able to reuse the existing building blocks of the model checker Storm. The approximation algorithm computes tight results for the reliability and MTTF within seconds. Thus, the approximation scales well for large scenarios with millions of states which can be analysed quickly by only building the most relevant fraction of the state space.

8.6.3 Related work

Earlier work [MS09] considers an automotive case study where functional blocks are translated to static fault trees without treating the partitioning on hardware architectures. [KL18] has a similar setting but focuses more on causal explanations and less on analysis performance of large-scale models. The evaluation of various options from the design space by a translation to fault trees, and applying fault tree analysis has also been considered for air traffic control [GCMT+16]. The effect of different topologies of a FlexRay bus has been assessed using FTA in [LCWC12] and identified the need for modelling dynamic aspects. The analysis of architecture decisions under safety aspects has been considered in e.g. [RBFA+14] using a dedicated description language and an analytical evaluation. Safety analysis for component-based systems has been considered in [GKP05], using state-event fault trees. Qualitative FTA has been used in [AOMM13] for ISO 26262 compliant evaluation of hardware. Different hardware partitionings are constructed and analysed using an Architecture Description Language (ADL) in [WRTP+13]. ADL-based dependability analysis has been investigated for several languages, e.g. AADL [BCKN+11], UML [LL11], Arcade [BCHK+08], and HiP-HOPS [CJLP+08]. These approaches typically have a steeper learning curve than the use of DFTs. The powerful Möbius analysis tool [CGKR+09] has been extended with dynamic reliability blocks [KS16a]. Model checking for safety analysis has been proposed by, e.g. [BCLM+15] which focuses on AltaRica, and does not cover probabilistic aspects.

Summary

- ➜ A vehicle guidance system is analysed through a model-based approach using DFTs.
- ➜ The system functions—given as a functional block diagram—are translated into a DFT representing the system failure in terms of function block failures. Each function block is automatically translated into a block DFT.
- ➜ The given mapping of function blocks onto the hardware architecture is automatically translated into connections between the block DFTs and hardware DFTs.
- ➜ The complete DFT models the overall system failure in terms of hardware failures, communication bus failures, internal failures, etc.
- ➜ The main benefit is the flexibility where new partitionings—of function blocks on hardware—and architectural changes can easily and automatically be integrated.
- ➜ The obtained fault trees are among the largest DFTs to date. DFTs with up to 100 BEs are analysed within minutes.
- ➜ The DFTs are not only analysed w.r.t. to common measures such as reliability and MTTF but also w.r.t. more complex measures taking degradation of components into account. Using probabilistic model checking allows to check a wide and varied range of measures out-of-the-box.

9 Railway Station Areas

Assumptions

The DFT is *conventional* and uses inclusive variants $\text{PAND}_{\leq}$, $\text{POR}_{\leq}$ and $\text{SEQ}_{\leq}$.

Our second case study stems from railway engineering. We consider infrastructure elements in railway station areas—e.g. switches, signals, etc.—and the effect of their failures on the routability of trains in the station area.

The content of this chapter was first presented in our publication 'A DFT Modeling Approach for Infrastructure Reliability Analysis of Railway Station Areas' [11]. Large parts of this chapter are taken verbatim from this publication.

9.1 Introduction

Strategic decisions in infrastructure design and asset management of railway networks are extremely critical as renewal cycles can easily span several decades and decisions tend to shape the network layout for decades. This is why a quantitative a-priori analysis of infrastructure reliability and performance is highly important.

Formal methods have found widespread application in verification of hard- and software standards in safety-critical applications such as interlockings [BCLP+15; CLSQ+17; Fan13; FMGF10; ITLR18] or train communications [BBC18; BCPV17; HJU05; PQ09]. The need for formal system analysis has recently been emphasized in the rail-specific CENELEC norms [EN50126; EN50128; EN50129; EN50159]. To date, formal method applications are mostly delimited to verifying specifications of individual components by original equipment manufacturers (OEMs), while on the network level heuristic or manual approaches continue to dominate [LJ18].

We present an infrastructure reliability model based on dynamic fault trees for the performance analysis and verification of railway station areas. The system description is based on train paths projected in the interlocking system, which are the fundamental element for train routing and operation in railway systems. Train path functionality is conditioned on the operability of tracks as well as field elements such as switches or signals, which are typically specified in interlocking tables, cf. e.g. [FMGF10]. Quality parameters of these systems or subsystems thereof are reported to the infrastructure manager by OEMs and can also often be deduced from network records.

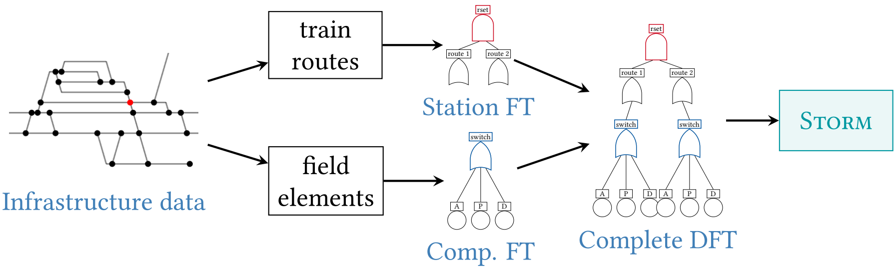


Figure 9.1: Overview of the DFT-based analysis approach for railway stations

Overview of approach | An overview of our fully automatic approach is depicted in Figure 9.1. From the given station infrastructure data, we derive the possible train routes in the railway station and the field elements, i.e. the physical infrastructure elements, such as switches, signals and track occupation detection systems. Similar to the approach for the vehicle guidance system—cf. Figure 8.1 on page 268—we build a two-level DFT. The station FT represents the possible train routes for each type of train. For each field element, we create a dedicated component FT representing the different failure types which can occur. We place special emphasis on field elements because they are known to be responsible for the majority of infrastructure failures in station and junction areas [BGDW18]. All fault trees are combined into one complete DFT which is analysed via Storm-dft. In our analysis, we use new performance metrics for asset management. The results can be used to gain a deeper understanding on the long term effects of strategic decisions in infrastructure design and maintenance planning.

Chapter structure | We shortly discuss the principles of railway operation procedures in Section 9.2. Section 9.3 presents our DFT-based modelling approach for railway networks on the system level. In Section 9.4, we discuss new performance metrics for quantitative analysis of the state of the infrastructure. The metrics become accessible through probabilistic model checking on the DFT model. We demonstrate the functionality and the capabilities of our approach in application scenarios for three German railway stations in Section 9.5. In Section 9.6, we review the current state of reliability analysis for railway networks and previous applications of formal methods.

9.2 Fundamentals of railway operations

Train operations in railway networks are based on interlocking systems, a combination of technical systems that prohibit conflicting movements of trains. In order to set a train route and to grant movement authority to a train, preconditions on the state of

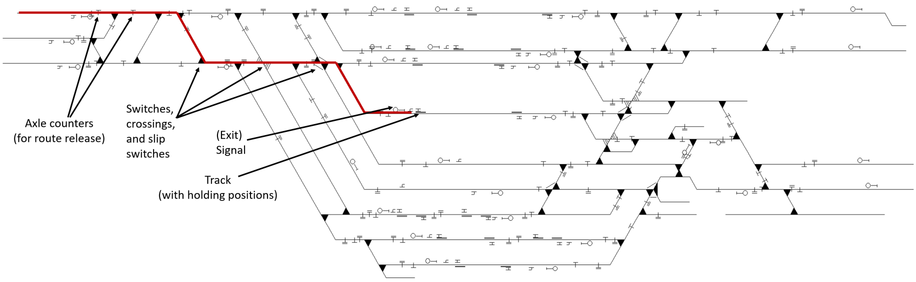


Figure 9.2: Railway station with exit route. The central station area (middle) is connected to three adjacent railway lines: two double-track lines (right) and a three-track line segment (left). Field elements included in the interlocking system such as axle counters, switches and signals are marked.

tracks and field elements have to be met, which are typically stored in interlocking control tables [FMGF10; WN18]. The wayside infrastructure consists of the permanent way including tracks, switches and crossings, and field elements required by the signalling and train control system such as signals and axle counters or track circuits to detect train movements. For radio-based train control systems, track-bound balises transmitting position indication and movement authority are required as well. In this work, we focus on elements in signal-based fixed-block train control, which remains the standard for train operations in stations (see Figure 9.2 for an example).

Similar to [WN18] and the verification of interlocking systems (cf. [Fan13]), our analysis is based on train routes. *Train routes* denote paths on the infrastructure which are delimited by signals governing station entry and exit. As no additional driving indication is given to train drivers en route in standard operations, the entire route has to be cleared and set for the train before the movement authority—by means of signals—is given. As a result, train routes can be seen as the fundamental routing elements in station areas and will be the base of our model. train route

9.3 DFT model for railway station reliability analysis

9.3.1 DFT model for railway station

The fault tree for the railway station area focuses on the possible train routing options. Different train types are associated with route sets, sets of possible train routes within the station area. Train routes in a route set are prioritised. The train route with the highest priority is the *standard route*. A *train route* consists of up to two train paths: ·standard route
a train path leading from the station entry to a halt in the station area, and similarly

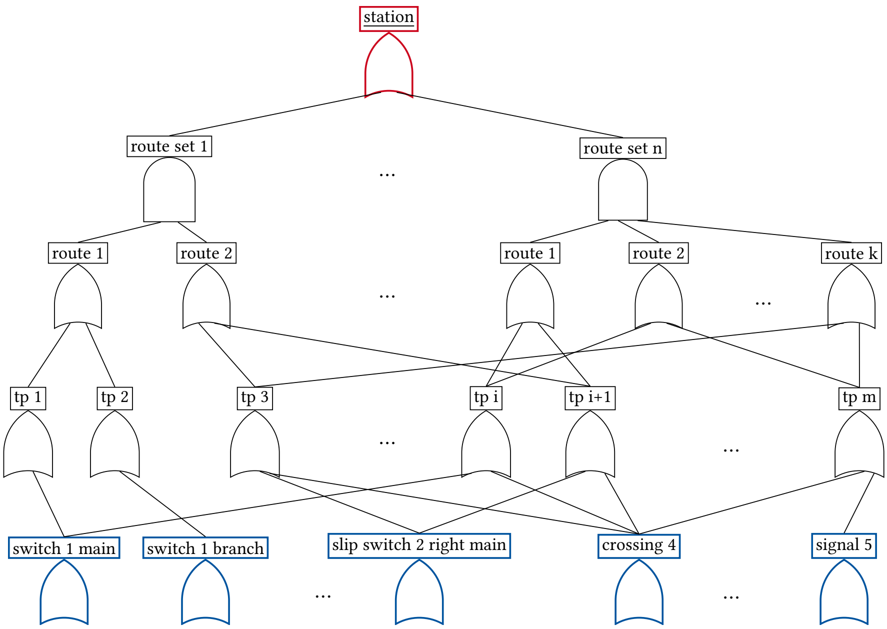


Figure 9.3: Railway station fault tree

a train path leading to the station exit. For trains starting or ending in the station, only one train path is present. A *train path* is specified in the interlocking system and determines, for example, the required state of switches and signals along the path.

train path

The DFT model for a railway station is given in Figure 9.3. A station is considered failed if no routing is possible any more for at least one train type. The routing for a train type is impossible if all train routes in the corresponding route set are unavailable. A train route is unavailable if at least one of the train paths tp is unavailable. A train path can be used in multiple train routes. Lastly, a train path tp is unavailable if at least one element (e.g. signal, crossing, etc.) along the path has failed. For switches and slip switches, the correct track of the component has to be unavailable. If, for example, a train path requires the main track of a switch and only the branch track is unavailable, then the train path is still available. Field elements can be used in multiple train paths.

9.3.2 DFT models for infrastructure components

In the following, we present DFT models for all relevant wayside infrastructure elements in the station area. A special focus lies on switches which are most important for routing, and are modelled with the greatest detail.

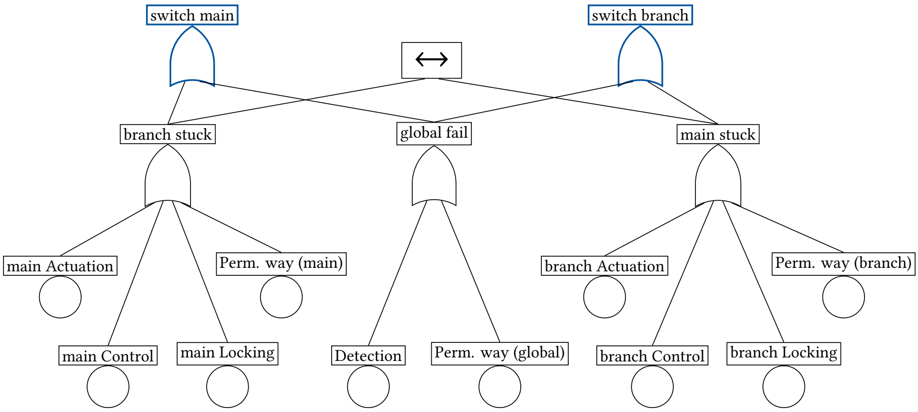


Figure 9.4: Switch fault tree

9.3.2.1 Switches

Switches allow routing on either the main track or the branching track depending on the current state of the blades. When changing the routing, the actuation and the control of the motor have to work in order to move the blades. Moreover, the locking mechanism has to work as well, to ensure that the blades are safely locked in their final position and do not move underneath a train. One important aspect of the switch is that a failure in one of the tracks might still allow routing on the other track. For example, if the actuation fails to move the blades, but they are still safely lockable in their present position, the current route can still be used. switch

The DFT modelling the different failure types of a switch is depicted in Figure 9.4. We consider the status of the main track and the branch track separately. Both of the corresponding OR-gates (switch main and switch branch) might have connections to other parts of the DFT.

For the modelling of the failure modes, we rely on the detailed categorisation of switch failures provided in [BGDW18]—for the UK network—where 5 technical categories have been identified:

- Actuation (A): failures in the track switching process (blade movement, lock actuation),
- Control/power (C): failures in control or power supply of switch subsystems,
- Detection (D): failure to detect/transmit the position of switch rails/locks,
- Locking (L): failure to lock the switch blades, and
- Permanent way (P): mechanical failures of rails, stretcher bars, slide chairs, etc.

Failures to detect the current switch position/locking (D) will render the entire switch

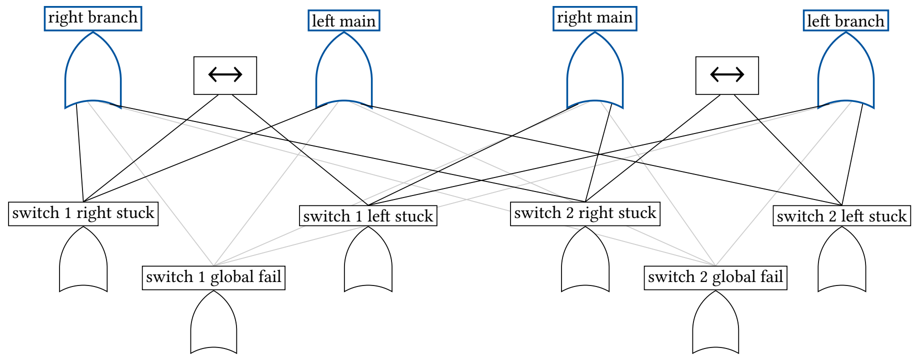


Figure 9.5: Slip switch fault tree (without BEs)

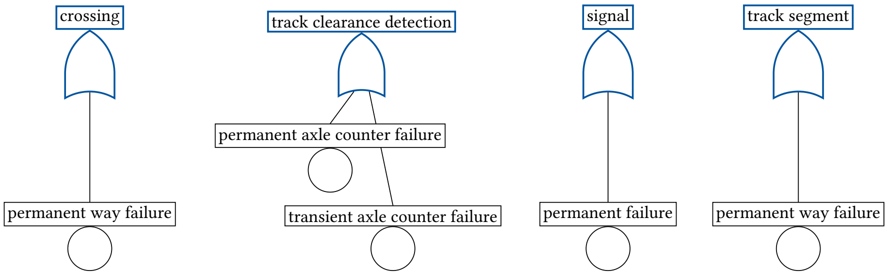
unusable. Failures of the permanent way (P) can be both position specific—blade rail, guiding rail failures—or global—ballast, crossing failures. Locking (L), Control (C) and Actuation (A) failures originate in the context of blade movements and typically only affect one of the two switch routing options. For instance, switch main is unavailable if branch stuck occurred, i.e. if the blades cannot be moved from their current (locked) position 'branch' to the 'main' position. We use a MUTEX to ensure that the switch can only be stuck in one position and not in both positions at the same time.

9.3.2.2 Slip switches

·slip switch

Slip switches allow up to four different routing options from two ingoing to two outgoing tracks. Intuitively, slip switches can be seen as a combination of two switches that each move the blades in two tracks. One motor sets the position of the blades on the two ingoing tracks—the right and left track. The other motor sets the position of the blades on the two outgoing tracks. Depending on the joint position of the blades, a train arriving on any ingoing track can be routed to the corresponding outgoing main or branch track. Not all routing options need to be technically realised in a slip switch.

The DFT model for a slip switch is depicted in Figure 9.5. It has four top-level elements—right/left × main/branch—which all might have connections to other DFT parts. Again, we distinguish between stuck branches and total failures. As the slip switch consists of two switches, if one of them has completely failed—switch 1 fail or switch 2 fail—all four routing options become unavailable and the slip switch has failed. However, if the blades for one switch are stuck, only two tracks become unavailable. These cases are represented by the four OR-gates with stuck. As before, two MUTEX ensure that the blades can only be stuck in one position.



(a) Crossing (b) Track clearance (c) Signal (d) Track segment

Figure 9.6: Fault trees for further components

For better visualisation, we omitted the BEs in the slip switch DFT, but they are similar as for the single switch DFT. For example, OR-gate switch 1 right stuck has four BEs for actuation, control, locking and permanent way failures.

9.3.2.3 Crossings

Crossings allow overlapping of two tracks without switching the tracks as in a slip switch. Without the presence of electromechanical components such as the motor or switch blades, the failure causes reduce to permanent way failures. The corresponding fault tree is depicted in Figure 9.6a.

9.3.2.4 Further components

As we focus on train routability in our DFT model, sub-FTs for switches have the richest topology. Further components such as signals or track clearance detection can be modelled as atomic components without detailed failure behaviour for routability. If desired, these components can of course be modelled in greater detail using all available gate types in DFTs. An overview of the fault trees for further components is given in Figure 9.6.

Track clearance detection | Track clearance detection reports whether the current track segment is occupied by a train. In Germany, axle counters are predominantly used for track clearance detection. Failures for axle counters (Figure 9.6b) can be subdivided into permanent failures of the component and transient failures, where a train axle was not detected and functionality is quickly restored by a reset [Kal17].

Signals | Signals failures (Figure 9.6c) are often confused with other system malfunctions connected with the interlocking system that prohibit the signal being switched

Table 9.1: Failure rates and parameters for switches

λ_A [1/h]	λ_C [1/h]	λ_D [1/h]	λ_L [1/h]	λ_P [1/h]	$\eta_{T,P}$
4.98×10^{-4}	2.26×10^{-4}	2.32×10^{-4}	1.28×10^{-4}	1.46×10^{-4}	0.11

to green. We consider intrinsic failures of the signal as a field element here. With the establishment of highly reliable LED technology they are typically not caused by the lamps, but by wayside electronics of the signal [Kal17].

Track segments | Track segments (Figure 9.6d) only experience failures due to wear from the trains running on the tracks. Failure rates for short track segments in stations, which are travelled at low speed, are typically small, cf. Section 9.3.3.

9.3.3 Failure rates

9.3.3.1 Switches and crossings

For the switch and crossing failure rates we rely on the extensive data from the UK Railway Network provided in [BGDW18]. For switches, we assume electromechanical actuation systems (Type HW/W63), which are the most widespread designs both in the UK and in the German railway network. Failure rates for the different failure causes are based on the data for mean time to failure requiring intervention (MTTFRI) in [BGDW18, Table 4].

Type (L), (C) and (A) failures in switches originate in the context of blade movements, only. As [Jal11] shows, almost 80 % of switch failure causes are due to blade obstruction—snow/ice or ballast—and insufficient lubrication of slide chairs. As these failures are more likely to occur when moving the switch blades to the position used less frequently, it could be argued for load dependent failure rates at this point. However, as no consistent information on the effects of load could be found in the literature, an even spread over the two branches has been assumed.

Permanent way (P) failures can yield both a complete failure of the switch in case elements used by all tracks are affected—crossings, ballast—or depend on the branch—guiding rails, blades. The share of permanent way failures that renders the entire switch unusable, regardless of the blade position, is denoted by $\eta_{T,P}$. We estimate $\eta_{T,P}$ based on the share of ballast, crossing, fishplate and sleeper failures in the UK failure cause data for switches provided in [Jal11, Table 1].

The failure rates and parameters are summarised in Table 9.1. Failure rates λ_A, λ_C, λ_D, λ_L and λ_P correspond to failure causes (A), (C), (D), (L) and (P), respectively. Data for crossings and slip switches rely on the same parameters. For crossings, only type (P)

Table 9.2: Failure rates [1/d] for track segments, signals, axle counters and track circuits.

Track Segments	Signals	Track Circuits	Axle Counters	
Failure	Failure	Failure	Reset request	Failure
4.4×10^{-4} (per km)	2.9×10^{-4}	5.5×10^{-4}	2.8×10^{-4}	1.1×10^{-4}

failures have to be considered. For slip switches type (A), (C), (D) and (L) failures apply to both switching motors, independently.

9.3.3.2 Further components

The failure rates for further components are given in Table 9.2. Track segments, track clearance detection and signals do not have degraded modes bearing on train routability. We therefore use aggregated values. For track segments and signals, failure rates consistent to the switch failure rates for the UK network can be estimated based on the number of reported failures and the approximate number of elements in the UK network[1]. The number of track circuits and axle counters in the UK could not be found. We therefore use failure data from [RGNS16] and [Kal17, Table 6].

9.4 Quality metrics for railway station areas

In the following, we present the metrics we compute in the context of railway station areas. The corresponding model-checking queries are formalised in *continuous stochastic logic (CSL)* (cf. Section 2.3.2) and given in Table 9.3.

General metrics | The standard metrics *unreliability* and *mean-time-to-failure (MTTF)* are computed as usual, cf. Section 5.3.1. Analysing the unreliability of the routing options is performed by computing the time-bounded reachability probability of the failure of a route or a train path.

Re-routing probability | The TLE represents the complete failure of at least one route set. To assess the probability that at least one train must be re-routed, we change the DFT model. The DFT is changed by only considering the standard route for each route set and removing all alternative train routes. The unreliability in the changed DFT corresponds to the re-routing probability. Another possibility—without the need to modify the DFT—is to calculate the reachability probability for the disjunction of failures of the standard routes.

[1] https://dataportal.orr.gov.uk/

Table 9.3: Model-checking queries

Measure	Model-checking query
Unreliability	$\mathbf{P}(\Diamond^{\leq t}\ \mathit{Failed})$
MTTF	$\mathbf{ET}(\Diamond\ \mathit{Failed})$
Unreliability for route i	$\mathbf{P}(\Diamond^{\leq t}\ \mathit{Failed}_{\text{route i}})$
Unreliability for train path i	$\mathbf{P}(\Diamond^{\leq t}\ \mathit{Failed}_{\text{tp i}})$
Re-routing probability	$\mathbf{P}(\Diamond^{\leq t}\ (\bigvee_{\text{standard route } i} \mathit{Failed}_{\text{route i}}))$
Criticality of element v	$\widetilde{I}^t_v$
Unrel. after component v failed	$\sum_{s \in S, \mathit{Failed}_v \in L(s)} \mathbf{P}(\neg \mathit{Failed}_v \cup s) \cdot \mathbf{P}_s(\Diamond^{\leq t}\ \mathit{Failed})$
MTTF after component v failed	$\sum_{s \in S, \mathit{Failed}_v \in L(s)} \mathbf{P}(\neg \mathit{Failed}_v \cup s) \cdot \mathbf{ET}_s(\Diamond\ \mathit{Failed})$

Criticality of infrastructure elements | One important metric in the station area model is the criticality of infrastructure elements, i.e. the influence of failures of a specific field element on the overall unreliability. We use the *Birnbaum importance index* [Bir68] to asses the criticality of components, cf. Section 5.3.2. Computing $\widetilde{I}^t_v$ for each element v gives insight into which element failures have the most impact on the routing in the station area.

Reliability until next maintenance | If an element has failed, it is important to know how fast the element should be repaired or replaced. We can compute the unreliability after component failure for a component v by looking at all states where v has already failed. Each state s should be the first instance where v has failed, i.e. v is operational in all predecessors of s. We compute the unreliability in each state s and multiply it with the probability to reach s in the first place. The time bound for the unreliability is chosen as a typical maintenance interval. In a similar fashion, the MTTF after component failure can be computed. Both computations use the improved algorithm from [8] which avoids performing the model checking query for each state and uses only two queries instead, cf. Section 8.4.2.

9.5 Application scenarios

9.5.1 Test cases

As application scenarios for our DFT-based station reliability model we consider three railway stations in the German state North Rhine Westphalia: Aachen Hbf, Mönchengladbach Hbf and Herzogenrath Bf. Aachen Hbf and Mönchengladbach Hbf are major stations with multiple starting and ending train lines, and 9 and 10 tracks, respectively—7 and 9 platform tracks, respectively. Herzogenrath is a considerably smaller medium size station with 4 platform tracks—3 in use—and a small freight yard.

In the following analysis, we focus on switches; track segments, axle counters and signals are not considered. While including the other elements is perfectly viable from a computational point of view, we focus on switches for three reasons:

1. Switches possess the most interesting routability properties as they exhibit various degraded modes which still allow the use in some train routes, depending on whether the branching or the main track of the switch is used.
2. Switch failures have been shown to be one of the most important factors in delay build-up [BGDW18]. As a result, more reliable switch designs have been discussed both by research [Col15] and OEMs, e.g. [Mor16]. There now exist multiple switch layouts to choose from, such that switches are interesting from an asset management and investment point of view as well. Failure rates of track segments, by contrast, can only be optimized to some extent by shortening inspection intervals, for instance.
3. Signal and axle counter failures tend to yield milder disruptions as fallback levels such as visual driving mode exist [WN18]. In addition, about 70 % of axle counter failures are detection errors only requiring a reset once track clearance has been confirmed [Kal17, Table 6]. Hence, the majority of axle counter failures are transient failures that can be resolved by train dispatchers.

9.5.2 Input data

For the input data we rely on infrastructure and train data for the German railway network, as specified in the XML-ISS and XML-KSS standards [BG03], data exchange formats for railway infrastructure and train data currently used by German infrastructure manager DB Netz AG. The format is related to the railML standard [NHSK04], to which our method could be adapted with minor modifications. As discussed before, the requirements of train routes with respect to the state of field elements are typically specified in control tables to which access is highly restrictive. We therefore follow a similar approach as the one recently discussed in [LJ18] and construct control tables from the infrastructure data. This approach can only provide an approximation to the actual situation specified in the interlocking control tables—which tend to vary between countries and even locally as a result of the exact track topology. However, it provides comparable data input, such that the approach is transferable to actual interlocking data.

Table 9.4: Considered scenarios

Scenario				Railway			
Id	Station	Variant	Max fail	#Route sets	#Routes	#Train paths	#Components
1	Aachen	std	∞	61	61	62	53
2		alt 5	4	23	115	41	54
3	Herzog.	std	∞	11	11	15	22
4		alt 5	4	9	19	15	24
5		alt 5	6	9	19	15	24
6	M'gladb.	std	∞	26	26	32	40
7		alt 5	4	11	43	25	41

9.5.3 Results and discussion

In our analysis, we use Storm-dft with all optimisations, cf. Chapter 5. In particular, we simplify the generated DFT models using rewriting, cf. Section 5.4.5. The complete workflow as presented in Figure 9.1 on page 296 is fully automatized. We run Storm-dft on an HP BL685C G7 restricted to 16 GB RAM and use a single 2.0 GHz core. The generated DFT models are available on our website[2].

Scenarios | The considered scenarios are given in Table 9.4. Each scenarios is identified by a unique id. For each of the three stations, we evaluate two different variants for the route sets: either a route set consists of only the standard route (std) or a route set contains the 5 most feasible routes according to priorities in the input data (alt 5). The fourth column in Table 9.4 indicates the maximal length of explored subsequent BE failures, i.e. we use the *depth* exploration heuristic, cf. Section 6.1.3. In our setting, it is reasonable to only consider a fixed number of subsequent failures as repairs or replacements would be performed after multiple failures. The last four columns in the table give the number of route sets, routes, train paths and components in the station.

The model characteristics for the considered scenarios are given in Table 9.5. The id in the first column references the scenario from Table 9.4. For the resulting DFT, we give the number of BEs, static gates and dynamic gates. The next two columns give the number of states and transitions in the resulting CTMCs. The last column states the time (in seconds) needed to build the CTMC.

Results | We analyse the resulting CTMCs according to the model checking queries specified in Section 9.4. The time bound is 90 days—a typical maintenance interval.

[2] http://www.stormchecker.org/publications/dfts-for-railway-stations

Table 9.5: Model characteristics

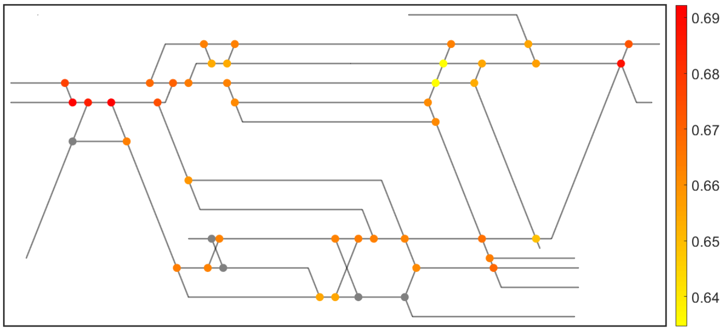
Id	DFT			CTMC		
	#BE	#Static	#Dynamic	#States	#Transitions	Build time [s]
1	544	459	54	2049	13 313	0.11
2	536	451	53	11 371 990	45 946 651	2006.16
3	194	137	19	257	1281	0.04
4	214	153	21	275 073	1 109 037	12.33
5	214	153	21	17 592 280	106 375 167	1110.48
6	480	325	48	8193	61 441	27.79
7	490	325	49	6 224 521	24 798 158	645.51

Table 9.6: Analysis results (MTTF in [d], computation times in [s])

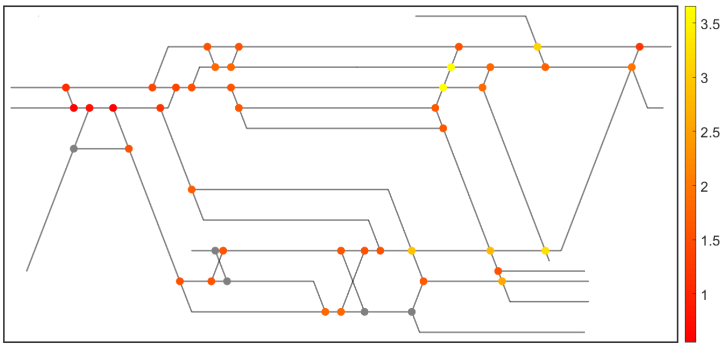
Id	General metrics			Unreliab. route		Criticality		MTTF after fail	
	Unrel.	MTTF	Time	Result	Avg. time	Result	Avg. time	Result	Avg. time
1	0.996	16.39	0.01	0.246	0.05	0.025	7.86	0.00	0.58
2	0.895	46.71	10.75	0.758	1.31	0.569	63.86	1.31	75.84
3	0.826	51.54	0.00	0.172	1.41	0.196	1.78	0.00	0.95
4	0.715	70.09	0.39	0.273	0.03	0.312	0.36	1.89	0.28
5	0.711	72.41	17.78	0.274	0.43	0.214	4.40	2.24	2.43
6	0.991	19.02	0.02	0.203	0.20	0.037	5.95	0.00	0.33
7	0.842	55.77	6.09	0.650	0.90	0.673	73.54	1.22	41.39

The results are presented in Table 9.6. The first column references the id. The next three columns give results for unreliability and the MTTF (in days), and the time (in seconds) needed to compute both metrics. The remainder presents the results for more intricate metrics: the unreliability of each route, the criticality of each switch and the MTTF after a switch failed. For all three metrics, we give one exemplary result and the average analysis time (in seconds). Note that we gain upper and lower bounds for all results on the partially explored models—because of limiting the number of subsequent failures—and present the worst-case results here.

Analysis results for all switches in Mönchengladbach Hbf (scenario 7) are depicted in Figure 9.7. The criticality results $\widetilde{I}_v^t$ are given in Figure 9.7a and the MTTF after the switch failed are given in Figure 9.7b. Each coloured dot represents a switch and its analysis result where red indicates a higher and yellow a lower value. Grey dots



(a) Criticality $\widetilde{I}_v^t$



(b) MTTF after component failure

Figure 9.7: Analysis of switches in Mönchengladbach Hbf for scenario 7

indicate switches which are not amongst the 5 train routes with highest priority for any train type.

Discussion | All models experience high unreliability within a 3-months period. Allowing multiple alternative paths instead of a standard path increases the reliability of the stations and the MTTF nearly triples for Aachen and Mönchengladbach.

When considering alternative paths, the state-space growth and therefore the building times become the main limiting factor, cf. Table 9.5. The time needed for model checking remains negligible even for the largest models though. For Herzogenrath we increased the maximal failure sequence from 4 to 6. While the state space greatly increases, the results only slightly change. Thus, it is reasonable to limit the number of considered subsequent failures and still get insightful results.

The criticality results in Figure 9.7a show that the switches on the ingoing and outgoing tracks are the most critical. Unavailable switches in these positions render large parts of the station area unavailable. It is also possible to see that the two yellow switches to the top right are not critical. This might indicate that most routes we considered are not using those tracks. The MTTF after switch failures in Figure 9.7b yield a similar result with subtle differences such as more distinct differences between the switches in the bottom right corner.

9.6 Related work

For a broad methodological overview on formal methods for railway software development, see [Bjø03]. The suitability of formal methods and tools for railway signalling and control applications with respect to the *safety integrity level (SIL)*—4 being the highest, 0 the lowest—has been reviewed in [Fan13]. In station areas, solid state interlocking programs (e.g. [ITLR18]) and railway signalling data (e.g. [IR16]) have been formally verified. In view of the migration of 'old' signal-based train control systems to new radio-based infrastructure such as ERTMS/ETCS (European Rail Traffic Management System / European Train Control System), verification of train control specifications—especially with respect to train radio communications—has received new attention recently. [CRT09] verifies consistency of ETCS requirements in a hybrid system setting including train dynamics using a combination of temporal logic with regular expressions. In [PQ09], ETCS is viewed as a hybrid system controller and controllability, safety, liveness and reactivity of ETCS in view of perturbations in train dynamics are investigated using deductive verification. [HJU05] presents a statechart extension and its transformation to discrete event simulation-based analysis of reliability in this context to verify compliance with QoS standards for train radio communication. [BCPV17] focuses on failure modelling of ERTMS/ETCS Level 3, an entirely radio-based control system standard. Stochastic Petri nets are used to provide a quantitative assessment of the effects of communication losses. In [BBC18], a Level 3 moving block signalling scenario is modelled and analysed with UPPAAL SMC.

Quality and safety requirements for electrotechnical systems in railway signalling applications have been specified in the CENELEC standards EN 50128 [EN50128], EN 50129 [EN50129] and EN 50159 [EN50159] EN 50128 recommends the use of formal methods even for software applications at SIL 1 and 2. The related EN 50126-1 standard [EN50126] provides guidelines for risk and asset management, also emphasizing the qualities of formal modelling approaches in this context.

To date, the use of formal methods in railway asset management remains limited and often focuses on specific elements or subsystems only. [Jal11] and [Kas17] discuss failure modes of tracks and switches and develop *Failure Mode and Effect Analysis*

9

(FMEA) schemes for asset management based on application scenarios from the British and Swedish railway networks. Markovian models for degradation modelling of rails and track foundations have been discussed in [PA13] and extended to a Petri net-based approach in [APR14].

Another class of models mentioned in EN 50126 are fault trees. Henry [Hen96] uses static fault trees to model train protection systems in a metro transit system. The railway power supply system and the effects of different maintenance strategies have been analysed using SFTs in [CHM07]. The model is solved using binary decision diagrams. [GKSL+14] discusses a DFT model for the investigation of general railway failure scenarios and maintenance strategies. Integrating new dynamic gates allows modelling of functional dependencies and spare parts typical for repair processes. Probabilistic model checking based on a Markov chain representation of the fault tree is applied to solve the model. An extension is provided in [RGNS16], where DFTs are used to optimise maintenance strategies for insulating joints, which are essential for train detection based on track circuits. A first step to transfer the use of fault trees to performability analysis of entire stations has been recently undertaken in [WN18], where the main focus is on the modelling of fallback levels in case of infrastructure disruptions.

Summary

- DFT-based model for the infrastructure in railway station areas.
- The focus is an field elements such as signals, axle counters and in particular switches, and the effect of their failures on the routability of trains in the station area.
- For a given railway area and its infrastructure data, the approach automatically extracts the relevant train paths for the train routing and constructs a corresponding DFT. This DFT is connected with dedicated DFTs for each infrastructure component along a train path.
- The complete DFT is analysed according to standard metrics such as reliability and MTTF, but also with respect to more complex metrics such as re-routing probabilities, criticality of components or MTTF after component failure.
- The evaluation shows that this approach for example can pinpoint the switches most critical for train operations in a railway station.

Part V

Conclusion

10 Conclusion and Outlook

10.1 Conclusion and Discussion

We set out to improve the existing state-of-the-art for dynamic fault trees with respect to three topics in particular (cf. Section 1.2): semantics, analysis and applications. We shortly summarise and discuss our contributions for each part. Concise summaries can also be found at the end of each chapter.

Semantics | We developed a generalised semantics for DFTs which captures the major semantics existing in the literature. Our formalisation via GSPNs (cf. Chapter 3) is convenient for comparison as differences between semantics can be characterized by the different priorities and different handling of non-determinism. Our investigation showed that in large parts, the semantics for each DFT gate are universally agreed upon. Minor differences are for example the treatment of simultaneous failures in priority gates which can be handled by introducing inclusive and exclusive variants. The main difference between semantics is the way in which failures propagate through a DFT—with respect to both time and causality. The proposed DFT semantics *NewGSPN* (cf. Section 4.5.1) captures the intuitively expected behaviour of failure propagation best. Furthermore, the provided generalised GSPN semantics provides a good basis for discussing DFT semantics and eventually agreeing on one standard[1].

In addition, the detailed investigation and enumeration of semantic intricacies for DFTs in Sections 4.1 and 4.2 provides a helpful guidance when working with DFTs. Being aware of these intricacies helps in creating better DFT models and also developing correct algorithms for DFT analysis. In particular, the common misconceptions that FDEPs can be replaced by ORs and SEQs by cold SPAREs were proven as incorrect in general. Note that the list of semantic questions is not exhaustive as new questions will certainly arise when combining DFT gates in uncommon ways.

Analysis | The analysis of DFTs is efficiently possible yielding accurate results by using probabilistic model checking. The crucial part however is to apply as many reduction techniques as possible during the state-space generation. We presented and implemented multiple optimisation techniques (cf. Section 5.4) such as using don't

[1] Although this might be a long process, cf. https://xkcd.com/927/

cares, exploiting symmetries, modularisation, detection of spurious non-determinism and graph rewriting. Combining these orthogonal optimisation techniques is crucial to significantly improve the analysis performance. The evaluation in Section 7.2 showed that our tool Storm-dft is state-of-the-art for DFT analysis.

In case all the optimisations do not lead to the desired performance improvement, the approximation approach via partial state-space generation (cf. Chapter 6) is a good alternative. The iterative nature of the approach allows to analyse DFTs in an interactive fashion where the user can dictate which states to explore next and when to stop. In particular for large DFTs, the partial state-space generation allows to mitigate the state-space explosion problem and yields precise enough results in reasonable time.

Lastly, we want to highlight that the good performance of Storm-dft—and even more so of its basis Storm—stems from countless hours of implementing ideas and algorithms, running tests and evaluating the performance, improving and revising existing code and algorithms, and supporting users and answering their questions. Storm-dft is publicly available at www.stormchecker.org and ready to be used for DFT analysis.

Applications | We presented two case studies using DFT models. The vehicle guidance system in Chapter 8 makes use of all types of DFT gates and the resulting DFTs are among the largest to date. The models for railway station areas in Chapter 9 are mostly static with the additional MUTEX-gates modelling dynamic behaviour. In both case studies, we utilise the hierarchical nature of fault trees by automatically generating different levels in the DFTs. Using fault trees allowed us to individually and independently model components which allows for great flexibility. New types of components can easily be integrated and existing ones can be adapted or exchanged without much effort. Another benefit of using DFTs is its graphical representation which allows to discuss the model without much previous knowledge. The different levels in a DFT further allow a customized view on the DFT, either on a high-level or focused on specific components.

The measures we analysed in both case studies showed the broad range of questions which can be answered via probabilistic model checking. Examples are calculating the criticality of switches in the railway station or estimating how long the vehicle guidance system can still be safely operated in degraded operation mode.

The evaluations also showed that our analysis techniques can be successfully applied to large DFT models within reasonable time. Further improvements on the performance are however still necessary to allow analysis of more complex systems.

10.2 Future work

We conclude this thesis by listing ideas and possible directions for future work.

10.2.1 Repairs in DFT

We excluded the modelling of repairs in this thesis (cf. Remark 1.1 on page 7). In most real-word systems however, repairs and maintenance actions are crucial to ensure continuously safe and reliable operation. Including repairs in fault trees is in principle a simple operation: BEs are additionally equipped with a repair rate which turns a failed BE operational again. While this simple approach works for SFTs, it quickly becomes very complex for DFTs.

Semantics | We already saw in Chapter 4 how many intricate semantic issues are present for non-repairable DFTs. Incorporating repairs leads to an even larger number of new questions such as:

- Does the dependent event of an FDEP get repaired if the trigger is repaired?
- Does the order of repairs influence the failure behaviour of priority gates?
- Can elements under a SEQ be repaired in any order?
- How are spare races resolved or avoided if a shared spare is repaired?

As a result, existing semantics for repairable DFTs [BC04; BCS10; MCCD⁺14; MBD20; BRS20] differ even more than non-repairable ones and no consensus can be seen.

Incorporating repairs in DFTs therefore requires a thorough investigation of the semantics. A first step could be to extend the DFT traces by an additional repair event and to define the behaviour of each gate on such traces. A second step would then be a thorough investigation of semantic issues when combining DFT gates. As seen from the short extract of question before, this step will require a significant amount of work.

Analysis | Incorporating repairs also poses new challenges for the DFT analysis. While the resulting Markov automata are acyclic for non-repairable DFTs, they become cyclic when allowing repairs. This does neither pose a problem for the state-space generation nor the model checking, but some of the optimisations must be adapted. In particular, don't cares cannot be exploited any more as the state of an element stays relevant even if its parents are failed. In an OR-gate for instance, all children must be repaired to render the OR operational again. Even if an OR is failed, we therefore must still track how many children are failed and don't cares cannot be used.

Support for repairs in DFTs therefore also requires the development of new state-space reduction techniques. On the plus side however, several techniques such as symmetries and modularisation can be used without large modification. The approximation via partial state-space generation should be applicable in its current form as well. The main modification is that the exploration cannot stop for failed states, because the system could still be repaired afterwards.

10.2.2 Extensions of DFT

DFTs are only one extension of SFTs. A possible direction of future work could be a combination of and inspiration by different SFT extensions.

Decoupling activation and claiming | A *Boolean-logic driven Markov decision process (BDMP)* [BB03] for example is very similar to a DFT. One interesting feature of BDMPs is that they perform activation via triggers, similar to how FDEPs propagate failures. Inspired from the activation behaviour in BDMPs, one could introduce an 'activation dependency' in DFTs which forwards the activation from an input element to an output element. This also allows to decouple the two behaviours present in SPARE-gates: activation and claiming. In addition to an activation dependency, a 'claiming gate' would be introduced which performs the claiming operation. The SPARE-gate is then a combination of both gates. The advantage of this approach is that activation can be performed without a SPARE and especially DFT Constraint 8 (Independence of modules) can be lifted. As a side-effect, the expressiveness of BDMPs compared to this new variant of DFTs could be determined conclusively.

Delayed failure propagation | Inspired by timed failure propagation graph models [AKMO09] and state/event fault trees (SEFTs) [KGF07], failure propagation with delays can be another extension of DFTs. Supporting delays allows to incorporate the timing behaviour of some events. One example is the occurrence of a fire which renders multiple components failed. This can presently be modelled with an FDEP. It would be more realistic however to add a deterministic delay to this FDEP as the fire does not immediately render the components failed. An advantage of having a 'delayed FDEP' would be that the difference between timed and causal order would be explicitly specified within a DFT. Thus, the order requirement for priority gates could be modelled more precisely.

10.2.3 Improved analysis

While the DFT analysis via Storm-dft is already in a good shape with several optimisations present, additional improvements are always beneficial.

BDD support | One improvement would be a faster analysis for largely static fault trees, i.e. DFTs where only a small part is actually dynamic—similar to static/dynamic fault trees (SD fault trees) [BBHK+16]. Analysing static subtrees via model checking is not efficient and dedicated methods should be used instead. A common approach is the use of binary decision diagrams (BDDs) [Rau93]. Future work could implement a BDD-based analysis approach in Storm-dft for improved performance on static (sub-)trees.

Compositional analysis | The state-space generation approaches of Storm-dft and DFTCalc [ABBG+13] are orthogonal. Storm-dft generates the complete state space in one process whereas DFTCalc follows a compositional approach [BCS10]: small input/output interactive Markov chains (I/O-IMC) for individual gates are composed and subsequently simplified before continuing the composition with other gates. While the composition introduces a large overhead for small- to medium-sized DFTs, it seems to offer benefits for large DFTs. A possible idea for future work is the combination of both approaches. First, Markov automata for small- to medium-sized DFTs are generated using Storm-dft. Afterwards, all resulting MAs are simplified and composed by following the approach of DFTCalc. For the composition to work, one either needs to convert an MA into an I/O-IMC or the composition needs to directly work on MAs. Note that this composition does not depend on independent subtrees—as required for modularisation—but is applicable in general. Thus, the composition could prove beneficial for highly coupled DFTs, such as the vehicle guidance systems from Chapter 8.

Part VI

Appendices

A Declaration of authorship

Eidesstattliche Erklärung

According to §12 (3) of the doctoral regulations of RWTH Aachen University, I am required to give the following declaration.

I, Matthias Volk,
declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research.
Hiermit erkläre ich an Eides statt / I do solemnely swear that:

1. This work was done wholly or mainly while in candidature for the doctoral degree at this faculty and university;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others or myself, this is always clearly attributed;
4. Where I have quoted from the work of others or myself, the source is always given. This thesis is entirely my own work, with the exception of such quotations;
5. I have acknowledged all major sources of assistance;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published before as indicated in Section 1.5 on page 14.

Matthias Volk, 16th April 2023, Aachen

B Bibliography

B.1 References

[AADL] SAE. '*Architecture, Analysis and Design Language AS5506*'. Standard. International Society for Automotive Engineers, 2004 (cited on page 12).

[ABBD+16] E. G. Amparore, G. Balbo, M. Beccuti, S. Donatelli and G. Franceschinis. '*30 Years of GreatSPN*'. *Principles of Performance and Reliability Modeling and Evaluation*. Springer Series in Reliability Engineering. Springer, 2016, pages 227–254. DOI: 10.1007/978-3-319-30599-8_9 (cited on pages 129, 155, 156, 240).

[ABBG+13] F. Arnold, A. Belinfante, F. van der Berg, D. Guck and M. Stoelinga. '*DFT-Calc: A Tool for Efficient Fault Tree Analysis*'. *SAFECOMP*. Volume 8153. Lecture Notes in Computer Science. Springer, 2013, pages 293–301. DOI: 10.1007/978-3-642-40793-2_27 (cited on pages 7, 155, 178, 239, 244, 319).

[ABHK18] P. Ashok, Y. Butkova, H. Hermanns and J. Kretínský. '*Continuous-Time Markov Decisions Based on Partial Exploration*'. *ATVA*. Volume 11138. Lecture Notes in Computer Science. Springer, 2018, pages 317–334. DOI: 10.1007/978-3-030-01090-4_19 (cited on page 229).

[ADH03] S. Amari, G. Dill and E. Howald. '*A new approach to solve dynamic fault trees*'. *RAMS*. 2003, pages 374–379. DOI: 10.1109/RAMS.2003.1182018 (cited on pages 191, 256).

[AHMS16] M. Ammar, G. B. Hamad, O. A. Mohamed and Y. Savaria. '*Efficient probabilistic fault tree analysis of safety critical systems via probabilistic model checking*'. *FDL*. IEEE, 2016, pages 1–8. DOI: 10.1109/FDL.2016.7880373 (cited on page 178).

[AKMO09] S. Abdelwahed, G. Karsai, N. Mahadevan and S. C. Ofsthun. '*Practical Implementation of Diagnosis Systems Using Timed Failure Propagation Graph Models*'. *IEEE Trans. Instrum. Meas.* 58.2 (2009), pages 240–247 (cited on page 318).

[AOMM13] N. Adler, S. Otten, M. Mohrhard and K. D. Müller-Glaser. '*Rapid safety evaluation of hardware architectural designs compliant with ISO 26262*'. *RSP*. IEEE, 2013, pages 66–72. DOI: 10.1109/RSP.2013.6683960 (cited on page 293).

[AP18] G. Agha and K. Palmskog. '*A Survey of Statistical Model Checking*'. *ACM Trans. Model. Comput. Simul.* 28.1 (2018), 6:1–6:39. DOI: 10.1145/3158668 (cited on page 268).

[APR14] J. D. Andrews, D. Prescott and F. D. Rozières. '*A stochastic model for railway track asset management*'. *Reliab. Eng. Syst. Saf.* 130 (2014), pages 76–84. DOI: 10.1016/j.ress.2014.04.021 (cited on page 310).

[AS98] A. Anand and A. Somani. '*Hierarchical analysis of fault trees with dependencies, using decomposition*'. *RAMS*. ISSN: 0149-144X. 1998, pages 69–75. DOI: 10.1109/RAMS.1998.653591 (cited on page 209).

[ASK09] A. Armoush, F. Salewski and S. Kowalewski. '*Design Pattern Representation for Safety-Critical Embedded Systems*'. *J. Softw. Eng. Appl.* 2.1 (2009), pages 1–12. DOI: 10.4236/jsea.2009.21001 (cited on page 271).

[BB03] M. Bouissou and J. Bon. '*A new formalism that combines advantages of fault-trees and Markov models: Boolean logic driven Markov processes*'. *Reliab. Eng. Syst. Saf.* 82.2 (2003), pages 149–163. DOI: 10.1016/S0951-8320(03)00143-1 (cited on pages 11, 202, 318).

[BBC18] D. Basile, M. H. ter Beek and V. Ciancia. '*Statistical Model Checking of a Moving Block Railway Signalling Scenario with Uppaal SMC - Experience and Outlook*'. *ISoLA (2)*. Volume 11245. Lecture Notes in Computer Science. Springer, 2018, pages 372–391. DOI: 10.1007/978-3-030-03421-4_24 (cited on pages 295, 309).

[BBHK+16] O. Bäckström, Y. Butkova, H. Hermanns, J. Krcál and P. Krcál. '*Effective Static and Dynamic Fault Tree Analysis*'. *SAFECOMP*. Volume 9922. Lecture Notes in Computer Science. Springer, 2016, pages 266–280. DOI: 10.1007/978-3-319-45477-1_21 (cited on pages 11, 318).

[BBMD+20] C. E. Budde, M. Biagi, R. E. Monti, P. R. D'Argenio and M. Stoelinga. '*Rare Event Simulation for Non-Markovian Repairable Fault Trees*'. *TACAS (1)*. Volume 12078. Lecture Notes in Computer Science. Springer, 2020, pages 463–482. DOI: 10.1007/978-3-030-45190-5_26 (cited on pages 156, 179, 219).

[BC04] A. Bobbio and D. Codetta-Raiteri. '*Parametric fault trees with dynamic gates and repair boxes*'. *RAMS*. 2004, pages 459–465. DOI: 10.1109/RAMS.2004.1285491 (cited on pages 172, 178, 208, 209, 211, 317).

[BCB89] S. C. Bruell, P. Chen and G. Balbo. '*Alternative Methods for Incorporating Non-exponential Distributions into Stochastic Timed Petri Nets*'. *PNPM.* IEEE Computer Society, 1989, pages 187–197. DOI: 10.1109/PNPM.1989.68552 (cited on page 67).

[BCCF+14] T. Brázdil, K. Chatterjee, M. Chmelik, V. Forejt, J. Kretínský, M. Z. Kwiatkowska, D. Parker and M. Ujma. '*Verification of Markov Decision Processes Using Learning Algorithms*'. *ATVA.* Volume 8837. Lecture Notes in Computer Science. Springer, 2014, pages 98–114. DOI: 10.1007/978-3-319-11936-6_8 (cited on page 229).

[BCHK+08] H. Boudali, P. Crouzen, B. R. Haverkort, M. Kuntz and M. Stoelinga. '*Architectural dependability evaluation with Arcade*'. *DSN.* IEEE Computer Society, 2008, pages 512–521. DOI: 10.1109/DSN.2008.4630122 (cited on page 293).

[BCKN+11] M. Bozzano, A. Cimatti, J. Katoen, V. Y. Nguyen, T. Noll and M. Roveri. '*Safety, Dependability and Performance Analysis of Extended AADL Models*'. *Comput. J.* 54.5 (2011), pages 754–775. DOI: 10.1093/comjnl/bxq024 (cited on pages 12, 293).

[BCLM+15] M. Bozzano, A. Cimatti, O. Lisagor, C. Mattarei, S. Mover, M. Roveri and S. Tonetta. '*Safety assessment of AltaRica models via symbolic model checking*'. *Sci. Comput. Program.* 98 (2015), pages 464–483. DOI: 10.1016/j.scico.2014.06.003 (cited on page 293).

[BCLP+15] S. Busard, Q. Cappart, C. Limbrée, C. Pecheur and P. Schaus. '*Verification of railway interlocking systems*'. *ESSS.* Volume 184. EPTCS. 2015, pages 19–31. DOI: 10.4204/EPTCS.184.2 (cited on page 295).

[BCPV17] M. Biagi, L. Carnevali, M. Paolieri and E. Vicario. '*Performability evaluation of the ERTMS/ETCS – Level 3*'. *Transportation Research Part C: Emerging Technologies* 82 (2017), pages 314–336. DOI: 10.1016/j.trc.2017.07.002 (cited on pages 295, 309).

[BCS07] H. Boudali, P. Crouzen and M. Stoelinga. '*Coral: a tool for compositional reliability and availability analysis*'. *ARTIST Workshop* (2007) (cited on page 155).

[BCS10] H. Boudali, P. Crouzen and M. Stoelinga. '*A Rigorous, Compositional, and Extensible Framework for Dynamic Fault Tree Analysis*'. *IEEE Trans. Dependable Secur. Comput.* 7.2 (2010), pages 128–143. DOI: 10.1109/TDSC.2009.45 (cited on pages 7, 9, 55, 87, 90, 105, 110, 123, 141, 145, 154, 155, 158, 168, 208, 212, 244, 317, 319).

B

[BD05a] H. Boudali and J. B. Dugan. '*A new Bayesian network approach to solve dynamic fault trees*'. *RAMS*. 2005, pages 451–456. DOI: 10.1109/RAMS.2005.1408404 (cited on pages 7, 9, 178, 219).

[BD05b] H. Boudali and J. B. Dugan. '*A discrete-time Bayesian network reliability modeling and analysis framework*'. *Reliab. Eng. Syst. Saf.* 87.3 (2005), pages 337–349. DOI: 10.1016/j.ress.2004.06.004 (cited on pages 172, 219, 255).

[BD06] H. Boudali and J. B. Dugan. '*A continuous-time Bayesian network reliability modeling, and analysis framework*'. *IEEE Trans. Reliab.* 55.1 (2006), pages 86–97. DOI: 10.1109/TR.2005.859228 (cited on page 172).

[BDHH+17] C. E. Budde, C. Dehnert, E. M. Hahn, A. Hartmanns, S. Junges and A. Turrini. '*JANI: Quantitative Model and Tool Interaction*'. *TACAS (2)*. Volume 10206. Lecture Notes in Computer Science. 2017, pages 151–168. DOI: 10.1007/978-3-662-54580-5_9 (cited on pages 129, 156).

[Ber05] D. P. Bertsekas. '*Dynamic programming and optimal control, 3rd Edition*'. Athena Scientific, 2005 (cited on page 155).

[Ber13] J. Bernoulli. '*Ars conjectandi, opus posthumum. Accedit Tractatus de seriebus infinitis, et epistola gallicé scripta de ludo pilae reticularis*'. Thurneysen Brothers, 1713 (cited on page 19).

[BG03] O. Brünger and T. Gröger. '*Fahrplantrassen managen und Fahrplanerstellung simulieren*'. *19. Verkehrswissenschaftliche Tage (VWT), Dresden, Germany*. 2003 (cited on page 305).

[BGDW18] S. D. Bemment, R. M. Goodall, R. Dixon and C. P. Ward. '*Improving the reliability and availability of railway track switching by analysing historical failure data and introducing functionally redundant subsystems*'. *Proceedings of the Institution of Mechanical Engineers, Part F: Journal of Rail and Rapid Transit* 232.5 (2018), pages 1407–1424. DOI: 10.1177/0954409717727879 (cited on pages 296, 299, 302, 305).

[BH02] H. C. Bohnenkamp and B. R. Haverkort. '*The Mean Value of the Maximum*'. *PAPM-PROBMIV*. Volume 2399. Lecture Notes in Computer Science. Springer, 2002, pages 37–56. DOI: 10.1007/3-540-45605-8_4 (cited on page 211).

[BHH19] Y. Butkova, A. Hartmanns and H. Hermanns. '*A Modest Approach to Modelling and Checking Markov Automata*'. *QEST*. Volume 11785. Lecture Notes in Computer Science. Springer, 2019, pages 52–69. DOI: 10.1007/978-3-030-30281-8_4 (cited on page 193).

[BHHH⁺13] C. Baier, E. M. Hahn, B. R. Haverkort, H. Hermanns and J. Katoen. '*Model checking for performability*'. *Math. Struct. Comput. Sci.* 23.4 (2013), pages 751–795. DOI: 10.1017/S0960129512000254 (cited on page 193).

[BHHK03] C. Baier, B. R. Haverkort, H. Hermanns and J. Katoen. '*Model-Checking Algorithms for Continuous-Time Markov Chains*'. *IEEE Trans. Software Eng.* 29.6 (2003), pages 524–541. DOI: 10.1109/TSE.2003.1205180 (cited on pages 44, 45, 180, 193, 211).

[BHHK15] Y. Butkova, H. Hatefi, H. Hermanns and J. Krcál. '*Optimal Continuous Time Markov Decisions*'. *ATVA*. Volume 9364. Lecture Notes in Computer Science. Springer, 2015, pages 166–182. DOI: 10.1007/978-3-319-24953-7_12 (cited on page 193).

[BHKK⁺21] C. E. Budde, A. Hartmanns, M. Klauck, J. Kretínský, D. Parker, T. Quatmann, A. Turrini and Z. Zhang. '*On Correctness, Precision, and Performance in Quantitative Verification: QComp 2020 Competition Report*'. *ISoLA*. Lecture Notes in Computer Science. Springer, 2021. to appear (cited on pages 9, 177, 180, 193, 212, 240).

[Bir68] Z. W. Birnbaum. '*On the importance of different components in a multicomponent system*'. Technical report TR-54. Laboratory of Statistical Research, University of Washington, 1968 (cited on pages 196, 304).

[Bjø03] D. Bjørner. '*New Results and Trends in Formal Techniques for the Development of Software for Transportation Systems*'. *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 2003 (cited on page 309).

[BK02] F. Bause and P. S. Kritzinger. '*Stochastic Petri nets - an introduction to the theory (2. ed.)*' Vieweg, 2002 (cited on page 52).

[BK08] C. Baier and J. Katoen. '*Principles of model checking*'. MIT Press, 2008 (cited on pages 9, 18, 45, 193, 200, 208, 211, 214, 255).

[BKHW05] C. Baier, J. Katoen, H. Hermanns and V. Wolf. '*Comparative branching-time semantics for Markov chains*'. *Inf. Comput.* 200.2 (2005), pages 149–214. DOI: 10.1016/j.ic.2005.03.001 (cited on page 200).

[BNS09] H. Boudali, A. P. Nijmeijer and M. Stoelinga. '*DFTSim: a simulation tool for extended dynamic fault trees*'. *SpringSim*. SCS/ACM, 2009 (cited on page 179).

[Bou07] M. Bouissou. '*A Generalization of Dynamic Fault Trees through Boolean logic Driven Markov Processes (BDMP)®*'. *ESREL*. 2007, page 7 (cited on page 11).

[Bou17] M. Bouissou. '*A Benchmark on Reliability of Complex Discrete Systems: Emergency Power Supply of a Nuclear Power Plant*'. *MARS@ETAPS*. Volume 244. EPTCS. 2017, pages 200–216. DOI: 10.4204/EPTCS.244.8 (cited on page 255).

[Boy91] M. A. Boyd. '*Dynamic fault tree models: Techniques for analysis of advanced fault tolerant computer systems*'. PhD thesis. Duke University, 1991 (cited on page 6).

[BRS20] C. E. Budde, E. Ruijters and M. Stoelinga. '*The Dynamic Fault Tree Rare Event Simulator*'. *QEST*. Volume 12289. Lecture Notes in Computer Science. Springer, 2020, pages 233–238. DOI: 10.1007/978-3-030-59854-9_17 (cited on pages 179, 239, 244, 317).

[Bud20] C. E. Budde. '*FIG: The Finite Improbability Generator*'. *TACAS (1)*. Volume 12078. Lecture Notes in Computer Science. Springer, 2020, pages 483–491. DOI: 10.1007/978-3-030-45190-5_27 (cited on page 156).

[Bun20] Bundesministeriums für Verkehr und digitale Infrastruktur. '*Pünktlichkeit und Zugausfälle bei der Deutschen Bahn bis 2019*'. Antwort of Kleine Anfrage, Drucksache 19/18864. 2020 (cited on page 3).

[CADC18] F. Chiacchio, J. I. Aizpurua, D. D'Urso and L. Compagno. '*Coherence region of the Priority-AND gate: Analytical and numerical examples*'. *Qual. Reliab. Eng. Int.* 34.1 (2018), pages 107–115. DOI: 10.1002/qre.2241 (cited on page 72).

[CAMN04] S. Cotton, E. Asarin, O. Maler and P. Niebert. '*Some Progress in Satisfiability Checking for Difference Logic*'. *FORMATS/FTRTFT*. Volume 3253. Lecture Notes in Computer Science. Springer, 2004, pages 263–276. DOI: 10.1007/978-3-540-30206-3_19 (cited on page 229).

[CEJS98] E. M. Clarke, E. A. Emerson, S. Jha and A. P. Sistla. '*Symmetry Reductions in Model Checking*'. *CAV*. Volume 1427. Lecture Notes in Computer Science. Springer, 1998, pages 147–158. DOI: 10.1007/BFb0028741 (cited on pages 203, 208).

[CGJL+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. '*Counterexample-Guided Abstraction Refinement*'. *CAV*. Volume 1855. Lecture Notes in Computer Science. Springer, 2000, pages 154–169. DOI: 10.1007/10722167_15 (cited on page 228).

[CGKR+09] T. Courtney, S. Gaonkar, K. Keefe, E. Rozier and W. H. Sanders. '*Möbius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models*'. *DSN.* IEEE Computer Society, 2009, pages 353–358. DOI: 10.1109/DSN.2009.5270318 (cited on page 293).

[CGL94] E. M. Clarke, O. Grumberg and D. E. Long. '*Model Checking and Abstraction*'. *ACM Trans. Program. Lang. Syst.* 16.5 (1994), pages 1512–1542. DOI: 10.1145/186025.186051 (cited on page 228).

[CHM07] S. K. Chen, T. K. Ho and B. H. Mao. '*Reliability evaluations of railway power supplies by fault-tree analysis*'. *IET Electric Power Applications* 1.2 (2007), pages 161–172. DOI: 10.1049/iet-epa:20060244 (cited on page 310).

[CJLP+08] D. Chen, R. Johansson, H. Lönn, Y. Papadopoulos, A. Sandberg, F. Törner and M. Törngren. '*Modelling Support for Design of Safety-Critical Automotive Embedded Systems*'. *SAFECOMP.* Volume 5219. Lecture Notes in Computer Science. Springer, 2008, pages 72–85. DOI: 10.1007/978-3-540-87698-4_9 (cited on page 293).

[CLSQ+17] Q. Cappart, C. Limbrée, P. Schaus, J. Quilbeuf, L. Traonouez and A. Legay. '*Verification of Interlocking Systems Using Statistical Model Checking*'. *HASE.* IEEE Computer Society, 2017, pages 61–68. DOI: 10.1109/HASE.2017.10 (cited on page 295).

[CM93] O. Coudert and J. C. Madre. '*Fault tree analysis: 10/sup 20/ prime implicants and beyond*'. *RAMS.* 1993, pages 240–245. DOI: 10.1109/RAMS.1993.296849 (cited on pages 5, 179).

[Col03] Columbia Accident Investigation Board. '*Report of Columbia Accident Investigation Board*'. 2003 (cited on page 3).

[Col15] I. Coleman. '*In2Rail Project Innovative Intelligent Rail, Deliverable D2.1 – Development of Novel S&C Motion/Locking Mechanisms: Design Concept Report*'. Technical report. Network Rail (NWR), 2015 (cited on page 305).

[CRT09] A. Cimatti, M. Roveri and S. Tonetta. '*Requirements Validation for Hybrid Systems*'. *CAV.* Volume 5643. Lecture Notes in Computer Science. Springer, 2009, pages 188–203. DOI: 10.1007/978-3-642-02658-4_17 (cited on page 309).

[CS98] D. Coppit and K. J. Sullivan. '*Formal Specification in Collaborative Design of Critical Software Tools*'. *HASE.* IEEE Computer Society, 1998, pages 13–20. DOI: 10.1109/HASE.1998.731590 (cited on pages 154, 159).

B

[CSD00] D. Coppit, K. J. Sullivan and J. B. Dugan. '*Formal Semantics for Computational Engineering: A Case Study on Dynamic Fault Trees*'. *ISSRE*. IEEE Computer Society, 2000, pages 270–282. DOI: 10.1109/ISSRE.2000.885878 (cited on pages 8, 55, 73, 105, 154, 160, 168).

[Daw04] C. Daws. '*Symbolic and Parametric Model Checking of Discrete-Time Markov Chains*'. *ICTAC*. Volume 3407. Lecture Notes in Computer Science. Springer, 2004, pages 280–294. DOI: 10.1007/978-3-540-31862-0_21 (cited on page 213).

[DBB90] J. B. Dugan, S. J. Bavuso and M. A. Boyd. '*Fault trees and sequence dependencies*'. *RAMS*. 1990, pages 286–293. DOI: 10.1109/ARMS.1990.67971 (cited on pages 6, 87, 110, 154).

[DBB92] J. B. Dugan, S. J. Bavuso and M. A. Boyd. '*Dynamic fault-tree models for fault-tolerant computer systems*'. *IEEE Transactions on Reliability* 41.3 (1992), pages 363–377. DOI: 10.1109/24.159800 (cited on pages 6, 7, 9, 26, 90, 110, 115, 145, 154, 177).

[DK05] P. R. D'Argenio and J. Katoen. '*A theory of stochastic systems part I: Stochastic automata*'. *Inf. Comput.* 203.1 (2005), pages 1–38. DOI: 10.1016/j.ic.2005.07.001 (cited on page 155).

[DM18] P. R. D'Argenio and R. E. Monti. '*Input/Output Stochastic Automata with Urgency: Confluence and Weak Determinism*'. *ICTAC*. Volume 11187. Lecture Notes in Computer Science. Springer, 2018, pages 132–152. DOI: 10.1007/978-3-030-02508-3_8 (cited on page 155).

[DR01] Y. Dutuit and A. Rauzy. '*Efficient algorithms to assess component and gate importance in fault tree analysis*'. *Reliab. Eng. Syst. Saf.* 72.2 (2001), pages 213–222. DOI: 10.1016/S0951-8320(01)00004-7 (cited on page 196).

[DR96] Y. Dutuit and A. Rauzy. '*A linear-time algorithm to find modules of fault trees*'. *IEEE Trans. Reliab.* 45.3 (1996), pages 422–425. DOI: 10.1109/24.537011 (cited on page 209).

[DTSG86] J. B. Dugan, K. S. Trivedi, M. K. Smotherman and R. M. Geist. '*The hybrid automated reliability predictor*'. *Journal of Guidance, Control, and Dynamics* 9.3 (1986), pages 319–331. DOI: 10.2514/3.20109 (cited on page 154).

[DVG97] J. B. Dugan, B. Venkataraman and R. Gulati. '*DIFtree: a software package for the analysis of dynamic fault tree models*'. *RAMS*. 1997, pages 64–70. DOI: 10.1109/RAMS.1997.571666 (cited on pages 7, 104, 154, 156, 279).

[EAHT19] Y. Elderhalli, W. Ahmad, O. Hasan and S. Tahar. '*Probabilistic Analysis of Dynamic Fault Trees using HOL Theorem Proving*'. *FLAP* 6.3 (2019), pages 469–512 (cited on pages 173, 217, 229).

[EC82] E. A. Emerson and E. M. Clarke. '*Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons*'. *Sci. Comput. Program.* 2.3 (1982), pages 241–266. DOI: 10.1016/0167-6423(83)90017-5 (cited on page 45).

[EHAT18] Y. Elderhalli, O. Hasan, W. Ahmad and S. Tahar. '*Formal Dynamic Fault Trees Analysis Using an Integration of Theorem Proving and Model Checking*'. *NFM*. Volume 10811. Lecture Notes in Computer Science. Springer, 2018, pages 139–156. DOI: 10.1007/978-3-319-77935-5_10 (cited on page 219).

[EHKZ13] C. Eisentraut, H. Hermanns, J. Katoen and L. Zhang. '*A Semantics for Every GSPN*'. *Petri Nets*. Volume 7927. Lecture Notes in Computer Science. Springer, 2013, pages 90–109. DOI: 10.1007/978-3-642-38697-8_6 (cited on pages 48, 51, 56, 156).

[EHZ10] C. Eisentraut, H. Hermanns and L. Zhang. '*On Probabilistic Automata in Continuous Time*'. *LICS*. IEEE Computer Society, 2010, pages 342–351. DOI: 10.1109/LICS.2010.41 (cited on pages 9, 42, 200).

[EN50126] CENELEC. '*EN 50126: Railway Applications – The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS)*'. Standard. European Committee for Electrotechnical Standardization, 2018 (cited on pages 295, 309).

[EN50128] CENELEC. '*EN 50128: Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems*'. Standard. European Committee for Electrotechnical Standardization, 2012 (cited on pages 295, 309).

[EN50129] CENELEC. '*EN 50129: Railway applications – Communication, signalling and processing systems – Safety related electronic systems for signalling*'. Standard. European Committee for Electrotechnical Standardization, 2017 (cited on pages 295, 309).

[EN50159] CENELEC. '*EN 50159: Railway applications – Communication, signalling and processing systems – Safety-related communication in transmission systems*'. Standard. European Committee for Electrotechnical Standardization, 2011 (cited on pages 295, 309).

[Eri99] C. A. Ericson. '*Fault Tree Analysis - A History*'. *Proceedings of the International Systems Safety Conference*. 1999, pages 1–9 (cited on page 4).

[Fan13] A. Fantechi. '*Twenty-Five Years of Formal Methods and Railways: What Next?*' *SEFM Workshops*. Volume 8368. Lecture Notes in Computer Science. Springer, 2013, pages 167–183. DOI: 10.1007/978-3-319-05032-4_13 (cited on pages 295, 297, 309).

[FAR76] J. Fussell, E. Aber and R. Rahl. '*On the Quantitative Analysis of Priority-AND Failure Logic*'. *IEEE Transactions on Reliability* R-25.5 (1976), pages 324–326. DOI: 10.1109/TR.1976.5220025 (cited on page 71).

[Fed20a] Federal Aviation Administration. '*Summary of the FAA's review of the Boeing 737 MAX*'. 2020 (cited on page 3).

[Fed20b] Federal Aviation Administration. '*System Safety Handbook*'. 2020 (cited on page 4).

[FGIM+02] G. Franceschinis, M. Gribaudo, M. Iacono, N. Mazzocca and V. Vittorini. '*DrawNET++: Model Objects to Support Performance Analysis and Simulation of Systems*'. *Computer Performance Evaluation / TOOLS*. Volume 2324. Lecture Notes in Computer Science. Springer, 2002, pages 233–238. DOI: 10.1007/3-540-46029-2_18 (cited on page 155).

[FLHD+16] M. Franz, C. T. Lopes, G. Huck, Y. Dong, S. O. Sümer and G. D. Bader. '*Cytoscape.js: a graph theory library for visualisation and analysis*'. *Bioinform.* 32.2 (2016), pages 309–311. DOI: 10.1093/bioinformatics/btv557 (cited on page 242).

[FMGF10] A. Ferrari, G. Magnani, D. Grasso and A. Fantechi. '*Model Checking Interlocking Control Tables*'. *FORMS/FORMAT*. Springer, 2010, pages 107–115. DOI: 10.1007/978-3-642-14261-1_11 (cited on pages 295, 297).

[GCMT+16] M. Gario, A. Cimatti, C. Mattarei, S. Tonetta and K. Y. Rozier. '*Model Checking at Scale: Automated Air Traffic Control Design Space Exploration*'. *CAV (2)*. Volume 9780. Lecture Notes in Computer Science. Springer, 2016, pages 3–22. DOI: 10.1007/978-3-319-41540-6_1 (cited on page 293).

[GD97] R. Gulati and J. B. Dugan. '*A modular approach for analyzing static and dynamic fault trees*'. *RAMS*. 1997, pages 57–63. DOI: 10.1109/RAMS.1997.571665 (cited on pages 7, 209, 211).

[GHHK+14] D. Guck, H. Hatefi, H. Hermanns, J. Katoen and M. Timmer. '*Analysis of Timed and Long-Run Objectives for Markov Automata*'. *Log. Methods Comput. Sci.* 10.3 (2014). DOI: 10.2168/LMCS-10(3:17)2014 (cited on pages 45, 46, 180, 193).

[GKP05] L. Grunske, B. Kaiser and Y. Papadopoulos. '*Model-Driven Safety Evaluation with State-Event-Based Component Failure Annotations*'. *CBSE*. Volume 3489. Lecture Notes in Computer Science. Springer, 2005, pages 33–48. DOI: 10.1007/11424529_3 (cited on page 293).

[GKSF16] M. Ghadhab, J. Kaienburg, M. Süßkraut and C. Fetzer. '*Is Software Coded Processing an Answer to the Execution Integrity Challenge of Current and Future Automotive Software-Intensive Applications?*' *Advanced Microsystems for Automotive Applications 2015*. Lecture Notes in Mobility. Springer, 2016, pages 263–275. DOI: 10.1007/978-3-319-20855-8_21 (cited on page 272).

[GKSL+14] D. Guck, J. Katoen, M. Stoelinga, T. Luiten and J. Romijn. '*Smart railroad maintenance engineering with stochastic model checking*'. *RAILWAYS*. Civil-Comp Proceedings. Civil-Comp Press, 2014, page 299. DOI: 10.4203/ccp.104.299 (cited on page 310).

[GLMS13] H. Garavel, F. Lang, R. Mateescu and W. Serwe. '*CADP 2011: a toolbox for the construction and analysis of distributed processes*'. *Int. J. Softw. Tools Technol. Transf.* 15.2 (2013), pages 89–107. DOI: 10.1007/s10009-012-0244-z (cited on page 244).

[Glu07] P. Gluchowski. '*Duration Calculus for Analysis of Fault Trees with Time Dependencies*'. *DepCoS-RELCOMEX*. IEEE Computer Society, 2007, pages 107–114. DOI: 10.1109/DEPCOS-RELCOMEX.2007.19 (cited on page 165).

[GLYZ+15a] D. Ge, M. Lin, Y. Yang, R. Zhang and Q. Chou. '*Quantitative analysis of dynamic fault trees using improved Sequential Binary Decision Diagrams*'. *Reliab. Eng. Syst. Saf.* 142 (2015), pages 289–299. DOI: 10.1016/j.ress.2015.06.001 (cited on pages 141, 255).

[GLYZ+15b] D. Ge, M. Lin, Y. Yang, R. Zhang and Q. Chou. '*Reliability analysis of complex dynamic fault trees based on an adapted K.D. Heidtmann algorithm*'. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability* (2015). DOI: 10.1177/1748006X15594694 (cited on page 7).

[GS18] E. Gascard and Z. Simeu-Abazi. '*Quantitative Analysis of Dynamic Fault Trees by means of Monte Carlo Simulations: Event-Driven Simulation Approach*'. *Reliab. Eng. Syst. Saf.* 180 (2018), pages 487–504. DOI: 10.1016/j.ress.2018.07.011 (cited on pages 136, 141).

[GS92] S. M. German and A. P. Sistla. '*Reasoning about Systems with Many Processes*'. *J. ACM* 39.3 (1992), pages 675–735. DOI: 10.1145/146637.146681 (cited on page 211).

[Hav95] B. R. Haverkort. '*In Search of Probability Mass: Probabilistic Evaluation of High-Level Specified Markov Models*'. *Comput. J.* 38.7 (1995), pages 521–529 (cited on page 229).

[Hen96] J. Henry. '*Automatic fault tree construction for railway safety systems*'. PhD thesis. Loughborough University, 1996 (cited on page 310).

[Her02] H. Hermanns. '*Interactive Markov Chains: The Quest for Quantified Quality*'. Volume 2428. Lecture Notes in Computer Science. Springer, 2002. DOI: 10.1007/3-540-45804-2 (cited on page 155).

[HH14] A. Hartmanns and H. Hermanns. '*The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification*'. *TACAS*. Volume 8413. Lecture Notes in Computer Science. Springer, 2014, pages 593–598. DOI: 10.1007/978-3-642-54862-8_51 (cited on page 9).

[HHHK+19] E. M. Hahn, A. Hartmanns, C. Hensel, M. Klauck, J. Klein, J. Kretínský, D. Parker, T. Quatmann, E. Ruijters and M. Steinmetz. '*The 2019 Comparison of Tools for the Analysis of Quantitative Formal Models - (QComp 2019 Competition Report)*'. *TACAS (3)*. Volume 11429. Lecture Notes in Computer Science. Springer, 2019, pages 69–92. DOI: 10.1007/978-3-030-17502-3_5 (cited on pages 9, 177, 193, 212, 240).

[Hix68] A. F. Hixenbaugh. '*Fault Tree for Safety*'. Section: Technical Reports. BOEING CO SEATTLE WA SUPPORT SYSTEMS ENGINEERING, 1968 (cited on page 4).

[HJU05] H. Hermanns, D. N. Jansen and Y. S. Usenko. '*From StoCharts to MoDeST: a comparative reliability analysis of train radio communications*'. *WOSP*. ACM, 2005, pages 13–23. DOI: 10.1145/1071021.1071023 (cited on pages 295, 309).

[HK00] H. Hermanns and J. Katoen. '*Automated compositional Markov chain generation for a plain-old telephone system*'. *Sci. Comput. Program.* 36.1 (2000), pages 97–127. DOI: 10.1016/S0167-6423(99)00019-2 (cited on page 211).

[HMPU+19] K. Höflinger, S. Müller, T. Peng, M. Ulmer, D. Lüdtke and A. Gerndt. '*Dynamic Fault Tree Analysis for a Distributed Onboard Computer*'. *2019 IEEE Aerospace Conference*. 2019 IEEE Aerospace Conference. 2019, pages 1–13. DOI: 10.1109/AERO.2019.8742128 (cited on page 7).

[IEC60812] IEC. '*IEC 60812: Failure modes and effects analysis* '. Standard. International Electrotechnical Commission, 1985 (cited on page 12).

[IEC61025] IEC. '*IEC 61025: Fault tree analysis*'. Standard. International Electrotechnical Commission, 1990 (cited on page 4).

[IEC61078] IEC. ‘*IEC 61078: Reliability block diagrams*’. Standard. International Electrotechnical Commission, 1991 (cited on page 12).

[Inv12] Investigation Committee on the Accident at Fukushima Nuclear Power Stations of Tokyo Electric Power Company. ‘*Final Report*’. 2012 (cited on page 3).

[IR16] A. Iliasov and A. B. Romanovsky. ‘*Formal Analysis of Railway Signalling Data*’. *HASE*. IEEE Computer Society, 2016, pages 70–77. DOI: 10.1109/HASE.2016.44 (cited on page 309).

[ISO11898] ISO. ‘*ISO 11898: Road vehicles – Controller area network (CAN)*’. Standard. International Organization for Standardization, 2015 (cited on page 280).

[ISO17458] ISO. ‘*ISO 17458: Road vehicles – FlexRay communications system*’. Standard. International Organization for Standardization, 2013 (cited on page 280).

[ISO26262] ISO. ‘*ISO 26262: Road vehicles – Functional safety*’. Standard. International Organization for Standardization, 2011 (cited on pages 4, 227, 267, 269, 272, 273, 279, 287, 291).

[ITLR18] A. Iliasov, D. Taylor, L. Laibinis and A. B. Romanovsky. ‘*Formal Verification of Signalling Programs with SafeCap*’. *SAFECOMP*. Volume 11093. Lecture Notes in Computer Science. Springer, 2018, pages 91–106. DOI: 10.1007/978-3-319-99130-6_7 (cited on pages 295, 309).

[Jal11] S. Jalili Hassankiadeh. ‘*Failure analysis of railway switches and crossings for the purpose of preventive maintenance.*’ Master thesis. KTH Royal Institute of Technology, 2011 (cited on pages 302, 309).

[JGKR+17] S. Junges, D. Guck, J. Katoen, A. Rensink and M. Stoelinga. ‘*Fault trees on a diet: automated reduction by graph rewriting*’. *Formal Aspects Comput.* 29.4 (2017), pages 651–703. DOI: 10.1007/s00165-016-0412-0 (cited on pages 10, 134, 137, 140, 173, 215, 217, 243, 269).

[JGKS16] S. Junges, D. Guck, J. Katoen and M. Stoelinga. ‘*Uncovering Dynamic Fault Trees*’. *DSN*. IEEE Computer Society, 2016, pages 299–310. DOI: 10.1109/DSN.2016.35 (cited on pages 8, 66, 110, 134, 141, 145, 149, 150, 172, 179, 237).

[JN07] F. V. Jensen and T. D. Nielsen. ‘*Bayesian Networks and Decision Graphs*’. 2nd edition. Information Science and Statistics. Springer, 2007. DOI: 10.1007/978-0-387-68282-2 (cited on page 172).

[Jun15] S. Junges. ‘*Simplifying Dynamic Fault Trees by Graph Rewriting*’. Master Thesis. RWTH Aachen University. 2015 (cited on pages 14, 22, 25, 56, 66, 104, 134–136, 141, 145, 146, 148, 149, 168, 173, 193, 215–217).

B

[Kab17] S. Kabir. '*An overview of fault tree analysis and its application in model based dependability analysis*'. *Expert Syst. Appl.* 77 (2017), pages 114–135. DOI: 10.1016/j.eswa.2017.01.058 (cited on page 141).

[Kal17] R. G. Kalvakunta. '*Reliability Modelling of ERTMS/ETCS*'. Master thesis. Norwegian University of Science and Technology, 2017 (cited on pages 301–303, 305).

[Kas17] E. Kassa. '*Analysis of failures within switches and crossings using failure modes and effects analysis methodology*'. *Proceedings of Intelliswitch Symposium.* 2017 (cited on page 309).

[KASP+20] S. Kabir, K. Aslansefat, I. Sorokos, Y. Papadopoulos and S. Konur. '*A Hybrid Modular Approach for Dynamic Fault Tree Analysis*'. *IEEE Access* 8 (2020), pages 97175–97188. DOI: 10.1109/ACCESS.2020.2996643 (cited on page 212).

[Kat16] J. Katoen. '*The Probabilistic Model Checking Landscape*'. *LICS.* ACM, 2016, pages 31–45. DOI: 10.1145/2933575.2934574 (cited on page 9).

[KGF07] B. Kaiser, C. Gramlich and M. Förster. '*State/event fault trees - A safety analysis model for software-controlled systems*'. *Reliab. Eng. Syst. Saf.* 92.11 (2007), pages 1521–1537. DOI: 10.1016/j.ress.2006.10.010 (cited on pages 11, 165, 318).

[KKB20] S. Khan, J. Katoen and M. Bouissou. '*Explaining Boolean-Logic Driven Markov Processes using GSPNs*'. *EDCC.* IEEE, 2020, pages 119–126. DOI: 10.1109/EDCC51268.2020.00028 (cited on page 11).

[KKNP01] J. Katoen, M. Z. Kwiatkowska, G. Norman and D. Parker. '*Faster and Symbolic CTMC Model Checking*'. *PAPM-PROBMIV.* Volume 2165. Lecture Notes in Computer Science. Springer, 2001, pages 23–38. DOI: 10.1007/3-540-44804-7_2 (cited on page 284).

[KL18] M. Kölbl and S. Leue. '*Automated Functional Safety Analysis of Automated Driving Systems*'. *FMICS.* Volume 11119. Lecture Notes in Computer Science. Springer, 2018, pages 35–51. DOI: 10.1007/978-3-030-00244-2_3 (cited on page 292).

[Kle99] T. A. Kletz. '*HAZOP and HAZAN: identifying and assessing process industry hazards*'. IChemE, 1999 (cited on page 12).

[KNP11] M. Z. Kwiatkowska, G. Norman and D. Parker. '*PRISM 4.0: Verification of Probabilistic Real-Time Systems*'. *CAV.* Volume 6806. Lecture Notes in Computer Science. Springer, 2011, pages 585–591. DOI: 10.1007/978-3-642-22110-1_47 (cited on pages 9, 178).

[KS16a] K. Keefe and W. H. Sanders. '*Reliability Analysis with Dynamic Reliability Block Diagrams in the Möbius Modeling Tool*'. *EAI Endorsed Trans. Security Safety* 3.10 (2016), e3. DOI: 10.4108/eai.14-12-2015.2262723 (cited on page 293).

[KS16b] D. Kroening and O. Strichman. '*Decision Procedures - An Algorithmic Point of View, Second Edition*'. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016. DOI: 10.1007/978-3-662-50497-0 (cited on pages 10, 229).

[KSYY20] Koorosh Aslansefat, Sohag Kabir, Youcef Gheraibia and Yiannis Papadopoulos. '*Dynamic Fault Tree Analysis: State-of-the-Art in Modeling, Analysis, and Tools*'. *Reliability Management and Engineering*. 1st edition. Reliability Management and Engineering. CRC Press, 2020, pages 73–112 (cited on pages 7, 178).

[Kwi03] M. Z. Kwiatkowska. '*Model checking for probability and time: from theory to practice*'. *LICS*. IEEE Computer Society, 2003, page 351. DOI: 10.1109/LICS.2003.1210075 (cited on page 9).

[LCWC12] K. Leu, J. Chen, C. Wey and Y. Chen. '*Generic Reliability Analysis for Safety-Critical FlexRay Drive-By-Wire Systems*'. *ICCVE*. IEEE Computer Society, 2012, pages 216–221. DOI: 10.1109/ICCVE.2012.48 (cited on page 293).

[LDB10] A. Legay, B. Delahaye and S. Bensalem. '*Statistical Model Checking: An Overview*'. *RV*. Volume 6418. Lecture Notes in Computer Science. Springer, 2010, pages 122–135. DOI: 10.1007/978-3-642-16612-9_11 (cited on page 268).

[LJ18] B. Luteberget and C. Johansen. '*Efficient verification of railway infrastructure designs against standard regulations*'. *Formal Methods Syst. Des.* 52.1 (2018), pages 1–32. DOI: 10.1007/s10703-017-0281-z (cited on pages 295, 305).

[LL11] F. Leitner-Fischer and S. Leue. '*QuantUM: Quantitative Safety Analysis of UML Models*'. *QAPL*. Volume 57. EPTCS. 2011, pages 16–30. DOI: 10.4204/EPTCS.57.2 (cited on page 293).

[LT87] N. A. Lynch and M. R. Tuttle. '*Hierarchical Correctness Proofs for Distributed Algorithms*'. *PODC*. ACM, 1987, pages 137–151. DOI: 10.1145/41840.41852 (cited on page 155).

[LXZL+07] D. Liu, W. Xing, C. Zhang, R. Li and H. Li. '*Cut Sequence Set Generation for Fault Tree Analysis*'. *ICESS*. Volume 4523. Lecture Notes in Computer Science. Springer, 2007, pages 592–603. DOI: 10.1007/978-3-540-72685-2_55 (cited on page 179).

B

[MB08] L. M. de Moura and N. Bjørner. '*Z3: An Efficient SMT Solver*'. *TACAS*. Volume 4963. Lecture Notes in Computer Science. Springer, 2008, pages 337–340. DOI: 10.1007/978-3-540-78800-3_24 (cited on pages 229, 236, 241).

[MBCD+95] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli and G. Franceschinis. '*Modelling with Generalized Stochastic Petri Nets*'. Volume 292. Wiley New York, 1995 (cited on pages 9, 48, 67).

[MBD20] R. E. Monti, C. E. Budde and P. R. D'Argenio. '*A compositional semantics for Repairable Fault Trees with general distributions*'. *LPAR*. Volume 73. EPiC Series in Computing. EasyChair, 2020, pages 354–372 (cited on pages 154–156, 168, 317).

[MC86] M. A. Marsan and G. Chiola. '*On Petri nets with deterministic and exponentially distributed firing times*'. *European Workshop on Applications and Theory of Petri Nets*. Volume 266. Lecture Notes in Computer Science. Springer, 1986, pages 132–145. DOI: 10.1007/3-540-18086-9_23 (cited on page 11).

[MCCD+14] G. Manno, F. Chiacchio, L. Compagno, D. D'Urso and N. Trapani. '*Conception of Repairable Dynamic Fault Trees and resolution by the use of RAATSS, a Matlab® toolbox based on the ATS formalism*'. *Reliab. Eng. Syst. Saf.* 121 (2014), pages 250–262. DOI: 10.1016/j.ress.2013.09.002 (cited on pages 179, 317).

[MCSD99] R. Manian, D. W. Coppit, K. J. Sullivan and J. B. Dugan. '*Bridging the gap between systems and dynamic fault tree models*'. *RAMS*. 1999, pages 105–111. DOI: 10.1109/RAMS.1999.744104 (cited on pages 154, 162, 164, 180).

[MDCS98] R. Manian, J. B. Dugan, D. Coppit and K. J. Sullivan. '*Combining Various Solution Techniques for Dynamic Fault Tree Analysis of Computer Systems*'. *HASE*. IEEE Computer Society, 1998, pages 21–28. DOI: 10.1109/HASE.1998.731591 (cited on pages 7, 157, 179).

[Mer10] G. Merle. '*Algebraic modelling of Dynamic Fault Trees, contribution to qualitative and quantitative analysis*'. PhD thesis. ENS Cachan, 2010 (cited on pages 173, 229).

[MMGN20] S. Müller, L. Mikaelyan, A. Gerndt and T. Noll. '*Synthesizing and optimizing FDIR recovery strategies from fault trees*'. *Science of Computer Programming* 196 (2020), pages 1–23. DOI: 10.1016/j.scico.2020.102478 (cited on page 92).

[Mo14] Y. Mo. '*A Multiple-Valued Decision-Diagram-Based Approach to Solve Dynamic Fault Trees*'. *IEEE Trans. Reliab.* 63.1 (2014), pages 81–93. DOI: 10.1109/TR.2014.2299674 (cited on page 179).

[Mog19] N. B. Moghadam. '*Algebraic Dynamic Fault Tree Analysis for Avionic Systems*'. Master Thesis. 2019 (cited on page 7).

[Mor16] S. Morant. '*New generation of turnouts promises to improve reliability and reduce costs*'. *IRJ International Rail Journal* 56.12 (2016), pages 40–41 (cited on page 305).

[MPB05] S. Montani, L. Portinale and A. Bobbio. '*Dynamic Bayesian Networks for modeling advanced Fault Tree features in dependability analysis*'. *ESREL*. AA Balkema Leiden, The Netherlands. 2005, pages 1415–1422 (cited on pages 7, 9, 113, 178).

[MPBR08] S. Montani, L. Portinale, A. Bobbio and D. C. Raiteri. '*Radyban: A tool for reliability analysis of dynamic fault trees through conversion into dynamic Bayesian networks*'. *Reliab. Eng. Syst. Saf.* 93.7 (2008), pages 922–932. DOI: 10.1016/j.ress.2007.03.013 (cited on page 178).

[MPBV+06] S. Montani, L. Portinale, A. Bobbio, M. Varesio and Codetta-Raiteri. '*A tool for automatically translating dynamic fault trees into dynamic Bayesian networks*'. *RAMS*. 2006, pages 434–441. DOI: 10.1109/RAMS.2006.1677413 (cited on pages 141, 172, 196, 219).

[MRL10] G. Merle, J.-M. Roussel and J.-J. Lesage. '*Improving the Efficiency of Dynamic Fault Tree Analysis by Considering Gate FDEP as Static*'. *European Safety and Reliability Conference (ESREL 2010).* Taylor & Francis, 2010, pp. 845–851 (cited on page 136).

[MRL14] G. Merle, J. Roussel and J. Lesage. '*Quantitative Analysis of Dynamic Fault Trees Based on the Structure Function*'. *Qual. Reliab. Eng. Int.* 30.1 (2014), pages 143–156. DOI: 10.1002/qre.1487 (cited on page 173).

[MRLB10] G. Merle, J.-M. Roussel, J.-J. Lesage and A. Bobbio. '*Probabilistic Algebraic Analysis of Fault Trees With Priority Dynamic Gates and Repeated Events*'. *IEEE Transactions on Reliability* 59.1 (2010), pages 250–261. DOI: 10.1109/TR.2009.2035793 (cited on page 141).

[MS09] M. L. McKelvin and A. Sangiovanni-Vincentelli. '*Fault Tree Analysis for the Design Exploration of Fault Tolerant Automotive Architectures*'. SAE Technical Paper 2009-01-1377. SAE International, 2009, pages 1–8. DOI: 10.4271/2009-01-1377 (cited on page 292).

[Neu10] M. R. Neuhäußer. '*Model checking nondeterministic and randomly timed systems*'. PhD thesis. RWTH Aachen University, 2010 (cited on page 45).

B

[NHSK04] A. Nash, D. Huerlimann, J. Schuette and V. P. Krauss. '*RailML – a standard data interface for railroad applications*'. *COMPRAIL*. Volume 74. WIT Press, 2004, pages 3–10. DOI: 10.2495/CR040241 (cited on page 305).

[Nor98] J. R. Norris. '*Markov chains*'. Cambridge series in statistical and probabilistic mathematics. Cambridge University Press, 1998 (cited on pages 9, 42, 44).

[OD03] Y. Ou and J. B. Dugan. '*Approximate sensitivity analysis for acyclic Markov reliability models*'. *IEEE Trans. Reliab.* 52.2 (2003), pages 220–230. DOI: 10.1109/TR.2003.809657 (cited on page 196).

[PA13] D. Prescott and J. Andrews. '*Modelling maintenance in railway infrastructure management*'. *RAMS*. 2013, pages 1–6. DOI: 10.1109/RAMS.2013.6517678 (cited on page 310).

[PQ09] A. Platzer and J. Quesel. '*European Train Control System: A Case Study in Formal Verification*'. *ICFEM*. Volume 5885. Lecture Notes in Computer Science. Springer, 2009, pages 246–265. DOI: 10.1007/978-3-642-10373-5_13 (cited on pages 295, 309).

[Pre86] Presidential Commission on the Space Shuttle Challenger Accident. '*Report of the Presidential Commission on the Space Shuttle Challenger Accident*'. 1986 (cited on page 3).

[Rai05] D. C. Raiteri. '*The Conversion of Dynamic Fault Trees to Stochastic Petri Nets, as a case of Graph Transformation*'. *Electron. Notes Theor. Comput. Sci.* 127.2 (2005), pages 45–60. DOI: 10.1016/j.entcs.2005.02.005 (cited on pages 26, 55, 56, 154, 155, 158, 159, 172).

[Rau11] A. Rauzy. '*Sequence Algebra, Sequence Decision Diagrams and Dynamic Fault Trees*'. *Reliab. Eng. Syst. Saf.* 96.7 (2011), pages 785–792. DOI: 10.1016/j.ress.2011.02.005 (cited on pages 173, 179).

[Rau93] A. Rauzy. '*New algorithms for fault trees analysis*'. *Reliability Engineering & System Safety* 40.3 (1993), pages 203–211. DOI: 10.1016/0951-8320(93)90060-C (cited on pages 5, 179, 212, 318).

[RB15] A. Rauzy and C. Blériot-Fabre. '*Towards a sound semantics for dynamic fault trees*'. *Reliab. Eng. Syst. Saf.* 142 (2015), pages 184–191. DOI: 10.1016/j.ress.2015.04.017 (cited on page 172).

[RBFA+14] V. Rupanov, C. Buckl, L. Fiege, M. Armbruster, A. C. Knoll and G. Spiegelberg. '*Employing early model-based safety evaluation to iteratively derive E/E architecture design*'. *Sci. Comput. Program.* 90 (2014), pages 161–179. DOI: 10.1016/j.scico.2013.10.005 (cited on page 293).

[RBNS+19] E. Ruijters, C. E. Budde, M. C. Nakhaee, M. Stoelinga, D. Bucur, D. Hiemstra and S. Schivo. '*Ffort: A Benchmark Suite for Fault Tree Analysis*'. *ESREL*. Research Publishing Services, 2019, pages 878–885. DOI: 10.3850/978-981-11-2724-3_0641-cd (cited on pages 10, 123, 239, 243, 285).

[Rei13] W. Reisig. '*Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*'. Springer, 2013 (cited on page 46).

[RGDP+16] E. Ruijters, D. Guck, P. Drolenga, M. Peters and M. Stoelinga. '*Maintenance Analysis and Optimization via Statistical Model Checking - Evaluating a Train Pneumatic Compressor*'. *QEST*. Volume 9826. Lecture Notes in Computer Science. Springer, 2016, pages 331–347. DOI: 10.1007/978-3-319-43425-4_22 (cited on page 7).

[RGNS16] E. Ruijters, D. Guck, M. van Noort and M. Stoelinga. '*Reliability-Centered Maintenance of the Electrically Insulated Railway Joint via Fault Tree Analysis: A Practical Experience Report*'. *DSN*. IEEE Computer Society, 2016, pages 662–669. DOI: 10.1109/DSN.2016.67 (cited on pages 7, 303, 310).

[RGRK+09] K. D. Rao, V. Gopika, V. V. S. S. Rao, H. S. Kushwaha, A. K. Verma and A. Srividya. '*Dynamic fault tree analysis using Monte Carlo simulation in probabilistic safety assessment*'. *Reliab. Eng. Syst. Saf.* 94.4 (2009), pages 872–883. DOI: 10.1016/j.ress.2008.09.007 (cited on pages 7, 9, 179, 219).

[RH03] M. Rausand and A. Hoyland. '*System Reliability Theory: Models, Statistical Methods, and Applications*'. 2nd edition. Wiley Series in Probability and Statistics. Wiley, 2003 (cited on page 12).

[RRBS19] E. Ruijters, D. Reijsbergen, P. de Boer and M. Stoelinga. '*Rare event simulation for dynamic fault trees*'. *Reliab. Eng. Syst. Saf.* 186 (2019), pages 220–231. DOI: 10.1016/j.ress.2019.02.004 (cited on pages 7, 9, 179, 219).

[RS15] E. Ruijters and M. Stoelinga. '*Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools*'. *Comput. Sci. Rev.* 15 (2015), pages 29–62. DOI: 10.1016/j.cosrev.2015.03.001 (cited on pages 4, 5, 7, 26, 110, 136, 141, 172, 178, 179, 211).

[Sch09] S. J. Schilling. '*Beitrag zur dynamischen Fehlerbaumanalyse ohne Modulbildungund zustandsbasierte Erweiterungen*'. PhD thesis. Bergische Universität Wuppertal, 2009 (cited on page 7).

B

[SDC99] K. J. Sullivan, J. B. Dugan and D. Coppit. '*The Galileo Fault Tree Analysis Tool*'. *FTCS.* IEEE Computer Society, 1999, pages 232–235. DOI: 10.1109/FTCS.1999.781056 (cited on pages 7, 154, 177, 178, 240).

[SL95] R. Segala and N. A. Lynch. '*Probabilistic Simulations for Probabilistic Processes*'. *Nord. J. Comput.* 2.2 (1995), pages 250–273 (cited on page 42).

[SN08] K. Slind and M. Norrish. '*A Brief Overview of HOL4*'. *TPHOLs.* Volume 5170. Lecture Notes in Computer Science. Springer, 2008, pages 28–32. DOI: 10.1007/978-3-540-71067-7_6 (cited on page 217).

[Spi89] J. M. Spivey. '*An introduction to Z and formal specifications*'. *Softw. Eng. J.* 4.1 (1989), pages 40–50. DOI: 10.1049/sej.1989.0006 (cited on page 154).

[Sta03] D. H. Stamatis. '*Failure mode and effect analysis: FMEA from theory to execution*'. Quality Press, 2003 (cited on page 12).

[Ste09] W. J. Stewart. '*Probability, Markov Chains, Queues, and Simulation*'. Princeton University Press, 2009 (cited on pages 9, 18, 32).

[Sto02] M. Stoelinga. '*An Introduction to Probabilistic Automata*'. *Bull. EATCS* 78 (2002), pages 176–198 (cited on page 42).

[SVDF+02] M. Stamatelatos, W. Vesely, J. Dugan, J. Fragola, J. Minarick and J. Railsback. '*Fault tree handbook with aerospace applications*' (2002) (cited on pages 4, 7, 136).

[TB17] K. S. Trivedi and A. Bobbio. '*Reliability and Availability Engineering - Modeling, Analysis, and Applications*'. Cambridge University Press, 2017 (cited on pages 4, 12, 20, 72, 179, 196, 270).

[TD04] Z. Tang and J. B. Dugan. '*Minimal cut set/sequence generation for dynamic fault trees*'. *RAMS.* 2004, pages 207–213. DOI: 10.1109/RAMS.2004.1285449 (cited on page 179).

[Tim13] M. Timmer. '*Efficient modelling, generation and analysis of Markov automata*'. PhD thesis. University of Twente, Enschede, Netherlands, 2013 (cited on pages 18, 42, 44, 45, 200).

[TPS13] M. Timmer, J. van de Pol and M. Stoelinga. '*Confluence Reduction for Markov Automata*'. *FORMATS.* Volume 8053. Lecture Notes in Computer Science. Springer, 2013, pages 243–257. DOI: 10.1007/978-3-642-40229-6_17 (cited on page 214).

[VGRH81] W. E. Vesely, F. F. Goldberg, N. H. Roberts and D. F. Haasl. '*Fault tree handbook*'. Technical report. Nuclear Regulatory Commission Washington DC, 1981 (cited on pages 4, 67).

[Wat61] H. Watson. '*Launch control safety study*'. *Bell Telephone Laboratories, Murray Hill, NJ USA* (1961) (cited on page 4).

[WN18] N. Weik and N. Nießen. '*Performability Analysis of Railway Systems*'. *2018 International Conference on Intelligent Rail Transportation (ICIRT)*. IEEE, 2018, pages 1–5. DOI: 10.1109/ICIRT.2018.8641589 (cited on pages 297, 305, 310).

[WP09] M. Walker and Y. Papadopoulos. '*Qualitative temporal analysis: Towards a full implementation of the Fault Tree Handbook*'. *Control Engineering Practice* 17.10 (2009), pages 1115–1125 (cited on pages 12, 71).

[WRTP+13] M. Walker, M. Reiser, S. Tucci Piergiovanni, Y. Papadopoulos, H. Lönn, C. Mraidha, D. Parker, D. Chen and D. Servat. '*Automatic optimisation of system architectures using EAST-ADL*'. *J. Syst. Softw.* 86.10 (2013), pages 2467–2487. DOI: 10.1016/j.jss.2013.04.001 (cited on page 293).

[XDM09] L. Xing, J. B. Dugan and B. A. Morrissette. '*An efficient approach to handling functional dependence loops*'. *ICRMS*. 2009, pages 37–40. DOI: 10.1109/ICRMS.2009.5270245 (cited on page 137).

[XTD12] L. Xing, O. Tannous and J. B. Dugan. '*Reliability Analysis of Nonrepairable Cold-Standby Systems Using Sequential Binary Decision Diagrams*'. *IEEE Trans. Syst. Man Cybern. Part A* 42.3 (2012), pages 715–726. DOI: 10.1109/TSMCA.2011.2170415 (cited on pages 179, 219).

[Yev16] O. Yevkin. '*An Efficient Approximate Markov Chain Method in Dynamic Fault Tree Analysis*'. *Qual. Reliab. Eng. Int.* 32.4 (2016), pages 1509–1520. DOI: 10.1002/qre.1861 (cited on pages 202, 228, 260).

[YY08] T. Yuge and S. Yanagi. '*Quantitative analysis of a fault tree with priority AND gates*'. *Reliab. Eng. Syst. Saf.* 93.11 (2008), pages 1577–1583. DOI: 10.1016/j.ress.2008.02.016 (cited on page 256).

[ZMFW09] X. Zhang, Q. Miao, X. Fan and D. Wang. '*Dynamic fault tree analysis based on Petri nets*'. *ICRMS*. 2009, pages 138–142. DOI: 10.1109/ICRMS.2009.5270223 (cited on pages 178, 219).

[ZXW20] S. Zhou, J. Xiang and W. E. Wong. '*Reliability analysis of dynamic fault trees with spare gates using conditional binary decision diagrams*'. *J. Syst. Softw.* 170 (2020), page 110766. DOI: 10.1016/j.jss.2020.110766 (cited on page 179).

[ZZDS01] H. Zhu, S. Zhou, J. B. Dugan and K. J. Sullivan. '*A benchmark for quantitative fault tree reliability analysis*'. *RAMS*. 2001, pages 86–93. DOI: 10.1109/RAMS.2001.902447 (cited on pages 208, 256).

B

[ZZLL11] H.-L. Zhang, C.-Y. Zhang, D. Liu and R. Li. '*A method of quantitative analysis for dynamic fault tree*'. *RAMS*. 2011, pages 1–6. DOI: 10.1109/RAMS.2011.5754471 (cited on page 136).

B.2 Co-authored Publications

Cited publications

[1] M. Volk, S. Junges and J. Katoen. '*Advancing Dynamic Fault Tree Analysis - Get Succinct State Spaces Fast and Synthesise Failure Rates*'. *SAFECOMP*. Volume 9922. Lecture Notes in Computer Science. Springer, 2016, pages 253–265. DOI: 10.1007/978-3-319-45477-1_20 (cited on pages 14, 178, 242–244, 248).

[2] C. Dehnert, S. Junges, J. Katoen and M. Volk. '*A Storm is Coming: A Modern Probabilistic Model Checker*'. *CAV (2)*. Volume 10427. Lecture Notes in Computer Science. Springer, 2017, pages 592–600. DOI: 10.1007/978-3-319-63390-9_31 (cited on pages 129, 155, 156, 239).

[3] M. Ghadhab, S. Junges, J. Katoen, M. Kuntz and M. Volk. '*Model-Based Safety Analysis for Vehicle Guidance Systems*'. *SAFECOMP*. Volume 10488. Lecture Notes in Computer Science. Springer, 2017, pages 3–19. DOI: 10.1007/978-3-319-66266-4_1 (cited on pages 15, 267).

[4] S. Junges, J. Katoen, M. Stoelinga and M. Volk. '*One Net Fits All - A Unifying Semantics of Dynamic Fault Trees Using GSPNs*'. *Petri Nets*. Volume 10877. Lecture Notes in Computer Science. Springer, 2018, pages 272–293. DOI: 10.1007/978-3-319-91268-4_14 (cited on pages 15, 56, 79, 80, 93, 106, 154, 156–158, 160).

[5] S. Junges, J. Katoen, M. Stoelinga and M. Volk. '*One Net Fits All: A unifying semantics of Dynamic Fault Trees using GSPNs*'. *CoRR* abs/1803.05376 (2018) (cited on page 56).

[6] M. Volk, S. Junges and J. Katoen. '*Fast Dynamic Fault Tree Analysis by Model Checking Techniques*'. *IEEE Trans. Ind. Informatics* 14.1 (2018), pages 370–379. DOI: 10.1109/TII.2017.2710316 (cited on pages 14, 154, 155, 178, 219, 230, 232, 233, 243, 244, 248).

[7] Y. Elderhalli, M. Volk, O. Hasan, J. Katoen and S. Tahar. '*Formal Verification of Rewriting Rules for Dynamic Fault Trees*'. *SEFM*. Volume 11724. Lecture Notes in Computer Science. Springer, 2019, pages 513–531. DOI: 10.1007/978-3-030-30446-1_27 (cited on page 217).

[8] M. Ghadhab, S. Junges, J. Katoen, M. Kuntz and M. Volk. '*Safety analysis for vehicle guidance systems with dynamic fault trees*'. *Reliab. Eng. Syst. Saf.* 186 (2019), pages 37–50. DOI: 10.1016/j.ress.2019.02.005 (cited on pages 15, 267, 304).

B

[9] S. Junges, E. Ábrahám, C. Hensel, N. Jansen, J. Katoen, T. Quatmann and M. Volk. '*Parameter Synthesis for Markov Models*'. *CoRR* abs/1903.07993 (2019) (cited on page 242).

[10] S. Khan, J. Katoen, M. Volk and M. Bouissou. '*Synergizing Reliability Modeling Languages: BDMPs without Repairs and DFTs*'. *PRDC*. IEEE, 2019, pages 266–275. DOI: 10.1109/PRDC47002.2019.00057 (cited on pages 11, 255).

[11] M. Volk, N. Weik, J. Katoen and N. Nießen. '*A DFT Modeling Approach for Infrastructure Reliability Analysis of Railway Station Areas*'. *FMICS*. Volume 11687. Lecture Notes in Computer Science. Springer, 2019, pages 40–58. DOI: 10.1007/978-3-030-27008-7_3 (cited on pages 15, 295).

[12] C. Hensel, S. Junges, J. Katoen, T. Quatmann and M. Volk. '*The Probabilistic Model Checker Storm*'. *CoRR* abs/2002.07080 (2020) (cited on pages 9, 10, 239, 268).

[13] S. Khan, J. Katoen, M. Volk and M. Bouissou. '*Scalable Reliability Analysis by Lazy Verification*'. *NFM*. Volume 12673. Lecture Notes in Computer Science. Springer, 2021, pages 180–197. DOI: 10.1007/978-3-030-76384-8_12 (cited on page 11).

Further peer-reviewed publications

N. Jansen, E. Ábrahám, M. Volk, R. Wimmer, J. Katoen and B. Becker. '*The COMICS Tool - Computing Minimal Counterexamples for DTMCs*'. *ATVA*. Volume 7561. Lecture Notes in Computer Science. Springer, 2012, pages 349–353. DOI: 10.1007/978-3-642-33386-6_27.

N. Jansen, F. Corzilius, M. Volk, R. Wimmer, E. Ábrahám, J. Katoen and B. Becker. '*Accelerating Parametric Probabilistic Verification*'. *QEST*. Volume 8657. Lecture Notes in Computer Science. Springer, 2014, pages 404–420. DOI: 10.1007/978-3-319-10696-0_31.

C. Dehnert, S. Junges, N. Jansen, F. Corzilius, M. Volk, H. Bruintjes, J. Katoen and E. Ábrahám. '*PROPhESY: A PRObabilistic ParamEter SYnthesis Tool*'. *CAV (1)*. Volume 9206. Lecture Notes in Computer Science. Springer, 2015, pages 214–231. DOI: 10.1007/978-3-319-21690-4_13.

C. Dehnert, S. Junges, N. Jansen, F. Corzilius, M. Volk, J. Katoen, E. Ábrahám and H. Bruintjes. '*Parameter Synthesis for Probabilistic Systems*'. *MBMV*. Albert-Ludwigs-Universität Freiburg, 2016, pages 72–74. DOI: 10.6094/UNIFR/10639.

S. Aflaki, M. Volk, B. Bonakdarpour, J. Katoen and A. Storjohann. '*Automated Fine Tuning of Probabilistic Self-Stabilizing Algorithms*'. *SRDS*. IEEE Computer Society, 2017, pages 94–103. DOI: 10.1109/SRDS.2017.22.

C Proofs

C.1 Proofs in Chapter 3

C.1.1 Proof of Lemma 3.18

We have to show Lemma 3.18 on page 130:

GSPN $\mathcal{G}_{\mathcal{F}}$ has at most $9 \cdot |V| \cdot \sigma_{max}$ places, $2 \cdot |\mathrm{BE}_{\mathcal{F}}|$ timed transitions and $6 \cdot |V| \cdot \sigma_{max}$ immediate transitions.

Proof. We give the precise sizes for each GSPN template in Table C.1 where $n = \sigma_{max}$. We give the number of interface places, the number of auxiliary places, the number of transitions—timed and immediate—and the number of arcs for each type of template. Note that the complete GSPN template for each DFT element consists of the specific template for the element type and the activation template.
The number of interface places is bounded by $|\mathcal{I}_{\mathcal{F}}| \leq 4 \cdot |V|$, because each DFT element $v \in \mathcal{F}$ has at most four corresponding interface places Failed_v, $\mathit{Unavail}_v$, Active_v and $\mathit{Disabled}_v$. The number of auxiliary places is bounded by $3 \cdot \sigma_{max} \cdot |V|$ with two additional places for the initialisation template. Thus, the total number of places is bounded by

$$\begin{aligned} |P| &\leq 4 \cdot |V| + 3 \cdot \sigma_{max} \cdot |V| + 2 \\ &= (4 + 3 \cdot \sigma_{max}) \cdot |V| + 1 \\ &\leq 9 \cdot \sigma_{max} \cdot |V|. \end{aligned}$$

The number of timed transitions is bounded by $|T_t| \leq 2 \cdot |\mathrm{BE}_{\mathcal{F}}|$. If no $\mathrm{CONST}_{\top}$ or $\mathrm{CONST}_{\bot}$ elements are present, the number is exactly $2 \cdot |\mathrm{BE}_{\mathcal{F}}|$. The number of immediate transitions is bounded by

$$\begin{aligned} |T_i| &\leq 4 \cdot \sigma_{max} \cdot |\mathrm{SPARE}_{\mathcal{F}}| + (2 \cdot \sigma_{max} + 2) \cdot |V \setminus \mathrm{SPARE}_{\mathcal{F}}| + 2 \\ &\leq (4 \cdot \sigma_{max} + 2) \cdot |V| \\ &\leq 6 \cdot \sigma_{max} \cdot |V|. \end{aligned}$$

Table C.1: Sizes for GSPN templates

Template type	# Interf. pl.	# Aux. pl.	# Transitions	# Arcs
SE_{exp}	4	0	2 timed	11
$\mathsf{CONST}_\top$	4	0	1 timed	4
$\mathsf{CONST}_\perp$	4	0	0	0
AND	3	0	1 immed.	$2n+3$
OR	3	0	n immed.	$5n$
VOT_k	3	$2n+1$	$n+1$ immed.	$4n+4$
$\mathsf{PAND}_\leq$	3	1	n immed.	$6n$
$\mathsf{PAND}_<$	3	$n-1$	n immed.	$7(n-1)+5$
$\mathsf{POR}_\leq$	3	1	n immed.	$4(n-1)+6$
$\mathsf{POR}_<$	3	0	1 immed.	$n+4$
SPARE	3	$3n$	$3n$ immed.	$14n+2$
SPARE (arb. claim.)	3	$2n+1$	$2n+1$ immed.	$12n+3$
FDEP	3	0	$n-1$ immed.	$6(n-1)$
PDEP_p	3	3	$n+2$ immed.	$6(n-1)+7$
SEQ	3	$n+1$	n immed.	$5(n-1)+3$
MUTEX	3	1	n immed.	n^2+2n
$\mathsf{templ}_{\mathsf{init}}$	0	1	1 immed.	2
Activation in modules	0	0	n immed.	$4n$
Activation in SPARE	0	0	n immed.	$6n$

The number of arcs is bounded by

$$
\begin{aligned}
&(\sigma_{max}^2 + 2\cdot\sigma_{max})\cdot|\mathsf{MUTEX}_{\mathcal{F}}| + (20\cdot\sigma_{max}+2)\cdot|V\setminus\mathsf{MUTEX}_{\mathcal{F}}| + 2\\
\leq&(\sigma_{max}^2 + 2\cdot\sigma_{max})\cdot|\mathsf{MUTEX}_{\mathcal{F}}| + (20\cdot\sigma_{max}+4)\cdot|V\setminus\mathsf{MUTEX}_{\mathcal{F}}|\\
\leq&(\sigma_{max}^2 \cdot|\mathsf{MUTEX}_{\mathcal{F}}| + (20\cdot\sigma_{max}+4)\cdot|V|.
\end{aligned}
$$

If no MUTEX is present in the DFT, the number of arcs is also linear in the number of DFT elements. Even if MUTEX are present, we can usually assume $\sigma_{max}\leq 20$.■

C.1.2 Proof of Lemma 3.19

We have to show Lemma 3.19 on page 130:

In GSPN $\mathcal{G}_{\mathcal{F}}$, for all interface places $p \in \mathcal{I} \setminus \{Disabled_v \mid v \in \mathsf{BE}_{\mathcal{F}}\}$, the tokens placed in p are never removed:

$$\forall m \in M : m(p) > 0 \implies m'(p) > 0,$$

where m' is a marking reachable from m.

Proof. Let $m \in M$ be a marking and $t \in \mathrm{enabled}(m)$ be a transition. Further let $p \in \{Failed_v, Active_v, Unavail_v \mid v \in \mathcal{F}\} = \mathcal{I} \setminus \{Disabled_v \mid v \in \mathsf{BE}_{\mathcal{F}}\}$ be a place. We need to show that for all markings m' with $m \xrightarrow{t} m'$ it holds:

$$m(p) \geq 1 \implies m'(p) \geq 1.$$

We define the set of places P' where each token which is removed from $p \in P'$ by a transition $t \in T$ is also put back to p by t:

$$I(t)(p) = 1 \implies O(t)(p) = 1.$$

We give P' for all templates:

- In $\mathsf{templ}_{\mathsf{VOT}_k}(v)$, it holds $P' = P \setminus \{Next_1, \ldots, Next_n, Collect\}$.
- In $\mathsf{templ}_{\mathsf{SPARE}}(v)$, it holds $P' = P \setminus \{Consider_1, \ldots, Consider_n\}$. For a SPARE with arbitrary claiming only place $Consider_v$ is excluded.
- In $\mathsf{templ}_{\mathsf{PDEP}_p}(v)$, it holds $P' = P \setminus \{Coin, Flip\}$.
- In $\mathsf{templ}_{\mathsf{SEQ}}(v)$, it holds

$$\begin{aligned} P' &= \left\{Failed_{\sigma(v)1}, \ldots, Failed_{\sigma(v)n}\right\} \\ &= P \setminus \left(\{Current_0, \ldots, Current_n\} \cup \left\{Disabled_{\sigma(v)1}, \ldots, Disabled_{\sigma(v)n}\right\}\right). \end{aligned}$$

- In $\mathsf{templ}_{\mathsf{MUTEX}}(v)$, it holds $P' = P \setminus \{Mutex\}$.
- In $\mathsf{templ}_{\mathsf{init}}$, it holds $P' = P \setminus \{Init\}$.
- For all other DFT types and the activation templates, we have $P' = P$.

In summary, for all gates we have

$$P' \supseteq \{Failed_v, Active_v, Unavail_v \mid v \in \mathcal{F}\}.$$

and thus, the claim holds. ∎

C

C.1.3 Proof of Lemma 3.21

We have to show Lemma 3.21 on page 130:

Each transition fires at most once.

Proof. We define a mapping $\mathsf{M} : T \to P$ where $p = \mathsf{M}(t)$ denotes a place which both:

- prevents firing t multiple times; formally, $O(t)(p) > 0$ and $H(t)(p) > 0$, and
- tokens placed in p are never removed; formally for all $t \in T$ and $m \xrightarrow{t} m'$: $m(p) > 0 \implies m'(p) > 0$.

Then, if t fires, it places a token in p and as $H(t)(p) > 0$ an immediate refiring of t is prohibited. Furthermore, as tokens are not removed from p, the transition can never fire a second time.

It remains to give the mapping M for each GSPN template in the following:

- For any transition $t \in \mathsf{templ}_{\mathsf{SE}_{exp}}(v) \cup \mathsf{templ}_{\mathsf{CONST}_\top}(v) \cup \mathsf{templ}_{\mathsf{AND}}(v) \cup \mathsf{templ}_{\mathsf{OR}}(v) \cup \mathsf{templ}_{\mathsf{POR}_<}(v)$ we set $\mathsf{M}(t) = \mathit{Failed}_v$.

- For any transition $t \in \mathsf{templ}_{\mathsf{PAND}_\leq}(v) \cup \mathsf{templ}_{\mathsf{POR}_\leq}(v)$ we set the mapping $\mathsf{M}(t) \in \{\mathit{FailSafe}_v, \mathit{Failed}_v\}$.

- For any transition $t \in \mathsf{templ}_{\mathsf{PAND}_<}(v)$ we set $\mathsf{M}(t) \in \{X_1, \dots, X_{n-1}, \mathit{Failed}_v\}$.

- For any transition $t \in \mathsf{templ}_{\mathsf{FDEP}}(v)$ we set $\mathsf{M}(t) \in \{\mathit{Failed}_{\sigma(v)_2}, \dots, \mathit{Failed}_{\sigma(v)_n}\}$.

- For any transition t in the activation templates, we set the mapping $\mathsf{M}(t) \in \{\mathit{Active}_{\sigma(v)_1}, \dots, \mathit{Active}_{\sigma(v)_n}, \mathit{Active}_v\}$.

- In $\mathsf{templ}_{\mathsf{VOT}_k}(v)$ we set $\mathsf{M}(\mathit{fail}) = \mathit{Failed}_v$. Firing one of the other transitions $\mathit{fail}_1, \dots, \mathit{fail}_n$ consumes the token in the corresponding place Next_i and thus, each transition is disabled afterwards.

- In $\mathsf{templ}_{\mathsf{SPARE}}(v)$ the token in $\mathit{Consider}_{\sigma(v)_i}$ either directly moves to $\mathit{Consider}_{\sigma(v)_{i+1}}$ (or to Failed_v in the end) or to $\mathit{Claimed}^v_{\sigma(v)_i}$. In both cases, *claim* and *unavail* are disabled afterwards. If a token is in $\mathit{Claimed}^v_{\sigma(v)_i}$ and transition *child-fail* fires, the token from $\mathit{Current}_i$ is consumed and *child-fail* is disabled afterwards. A similar argument holds for the SPARE with arbitrary claiming.

- In $\mathsf{templ}_{\mathsf{PDEP}_p}(v)$ transition *coin* consumes the token in *Coin* and cannot fire afterwards. Similarly, transitions *flip-success* and *flip-fail* consume the token in *Coin* and are disabled afterwards. Furthermore, both transitions are only enabled through the firing of *coin* which can fire only once. Thus, both transitions are only enabled once as well. For the remaining transitions, we set $\mathsf{M}(t) \in \{Failed_{\sigma(v)_2}, \ldots, Failed_{\sigma(v)_n}\}$.

- In $\mathsf{templ}_{\mathsf{SEQ}}(v)$ the transitions $next_1, \ldots, next_n$ can only fire from left to right moving the token from $Current_0$ to $Current_n$. Each transition $next_i$ removes the token from $Current_{i-1}$ and places it in $Current_i$. Therefore, $next_i$ is disabled.

- In $\mathsf{templ}_{\mathsf{MUTEX}}(v)$, firing of a transition consumes the token in *Mutex*. As a result, all transitions are disabled afterwards.

- In $\mathsf{templ}_{\mathsf{init}}$ transition *init* removes the token from *Init* and afterwards this transition is disabled. ■

C.1.4 Proof of Lemma 3.23

We have to show Lemma 3.23 on page 131:

The GSPN $\mathcal{G}_{\mathcal{F}}$ is bounded by

$$\max\{2, Collect_{max}, Disabled_{max}\}.$$

All places $P \setminus (\{Unavail_v \mid v \in \mathcal{F}\} \cup \{Collect \mid v \in \mathsf{VOT}_k\} \cup \{Disabled_v \mid v \in \mathsf{BE}\})$ are 1-bounded.

Proof. All places except $Unavail_v$, *Collect* and $Disabled_v$ are 1-bounded. Let $p \in \mathsf{M}(t)$ (for some t) where M is as in the proof for Lemma 3.21. A new token is only placed in p if there was no token before. This is ensured by the inhibitor arcs as previously explained. We consider all templates in detail:

- In each of $\mathsf{templ}_{\mathsf{SE}_{exp}}(v)$, $\mathsf{templ}_{\mathsf{CONST}_{\top}}(v)$, $\mathsf{templ}_{\mathsf{AND}}(v)$, $\mathsf{templ}_{\mathsf{OR}}(v)$, $\mathsf{templ}_{\mathsf{PAND}_{\leq}}(v)$, $\mathsf{templ}_{\mathsf{POR}_{\leq}}(v)$, $\mathsf{templ}_{\mathsf{POR}_{<}}(v)$ and $\mathsf{templ}_{\mathsf{MUTEX}}(v)$ at most one transition can be fired and places a token in $p \in \mathsf{M}(t)$. Thus all places $p \neq Unavail_v$ are 1-bounded.

- In $\mathsf{templ}_{\mathsf{VOT}_k}(v)$ place *Collect* can contain up to $|\sigma(v)|$ many tokens—one for each $fail_i$. All other places can contain only one token.

- In $\mathsf{templ}_{\mathsf{PAND}_{<}}(v)$ the transitions only fire from left to right, while adding tokens in X_1 to X_{n-1} and finally placing it in $Failed_v$ if there was none before.

- In $\mathsf{templ}_{\mathsf{SPARE}}(v)$ the token moves from left to right through the auxiliary places $Consider_{\sigma(v)_i}$. Because of the places $Current_i$, transitions *child-fail* can fire only once and at most one token can be placed in $Consider_{\sigma(v)_i}$. The single moving token ends up in $Failed_v$. There is no inhibitor arc for $Failed_v$ but due to the moving token, only one token is placed in $Failed_v$. Furthermore, a token is placed in $Unavail_{\sigma(v)_i}$ only if there was none before. However, if first a token is placed in $Unavail_{\sigma(v)_i}$ in the SPARE template, a second token can be added if $\sigma(v)_i$ fails—in the corresponding template—and tokens are added to $Failed_{\sigma(v)_i}$ and $Unavail_{\sigma(v)_i}$. Thus, $Unavail_v$ is 2-bounded. The boundedness of the SPARE with arbitrary claiming follows similar arguments.

- In the activation templates a token can only be placed in $Active_{\sigma(v)_1}, \ldots, Active_{\sigma(v)_n}$ and $Active_v$ if there was none before.
- In $\mathsf{templ}_{\mathsf{FDEP}}(v)$ all transitions can only place a token in $Failed_{\sigma(v)_2}, \ldots, Failed_{\sigma(v)_n}$ if there was none before.
- In $\mathsf{templ}_{\mathsf{PDEP}}(v)$ the token moves from *Coin* over *Flip* to *Forward*. Thus, each of the three places contains at most one token. All other places follow the reasoning of $\mathsf{templ}_{\mathsf{FDEP}}(v)$.
- In $\mathsf{templ}_{\mathsf{SEQ}}(v)$ the token moves through the places $Current_i$. A place $Disabled_{\sigma(v)_i}$ can initially contain a token for each parent SEQ. Thus, each $Disabled_v$ is bounded by $|\rho^{\mathsf{SEQ}}(v)|$.
- In $\mathsf{templ}_{\mathsf{init}}$ the first transition puts a token in $Active_{\top}$. As no *Active* place initially contains a token, the place contains at most one token. ■

C.2 Proofs in Chapter 5

C.2.1 Proof of Lemma 5.5

We have to show Lemma 5.5 on page 191:

Let $\mathcal{M}_{\mathcal{F}}$ be the MA for DFT $\mathcal{F}$. The state-space size of $\mathcal{M}_{\mathcal{F}}$ is in the worst case factorial in the number of BEs and PDEPs of $\mathcal{F}$:

$$|S| \leq (|\mathsf{BE}_{\mathcal{F}}| + 2 \cdot |\mathsf{PDEP}_{\mathcal{F}}| + 1)!,$$

where $n!$ is the factorial of n.

Proof. The state space of $\mathcal{M}_{\mathcal{F}}$ is generated by considering all possible traces of BE failures and evaluations of PDEPs. We divide the state space into 'levels' corresponding to the number of BEs and PDEPs which have failed already. In level zero, we only have the initial state where no BE is failed. In level one, one BE has failed or one PDEP was evaluated. We have $|\mathrm{BE}_{\mathcal{F}}|$ states corresponding to BE failures in this level. Additionally, PDEPs could have been evaluated in this level as well. For each PDEP we get two additional states: one for the successful coin flip and one for the unsuccessful one. In total, we get $n = |\mathrm{BE}_{\mathcal{F}}| + 2 \cdot |\mathrm{PDEP}_{\mathcal{F}}|$ states. From each state in level one, one of the remaining BEs or PDEPs can fail. We therefore have at most $(n-1) \cdot n$ states in level two. Continuing this construction, the total number of states is bounded by

$$|S| \leq 1 + n + n(n-1) + n(n-1)(n-2) + \cdots + n! = \sum_{i=0}^{n} \frac{n!}{(n-i)!}.$$

We can estimate this bound as follows:

$$\sum_{i=0}^{n} \frac{n!}{(n-i)!} = n! \sum_{i=0}^{n} \frac{1}{(n-i)!} = n! \sum_{k=0}^{n} \frac{1}{k!} \leq n! \cdot (n+1) = (n+1)!.$$

The state space size is therefore factorial in the number of BEs and PDEPs, i.e. $|S| \in \mathcal{O}\left((|\mathrm{BE}_{\mathcal{F}} \cup \mathrm{PDEP}_{\mathcal{F}}|)!\right)$. ■

C

C.3 Proofs in Chapter 6

C.3.1 Proof of Lemma 6.1

We have to show Lemma 6.1 on page 226:

Let D be a set of exponential distributions and let Λ denote the multi-set[1] of rates of distributions in D. We denote the multi-sets of Λ with cardinality i by

$$[\Lambda]^i = \{B \subseteq \Lambda \mid |B| = i\}.$$

The *expected time of the maximum of exponential distributions D* is given by

·maximum of exponential distributions

$$\mathbf{E}\big(\max(D)\big) = \mathbf{E}\big(\max(\Lambda)\big) = \sum_{i=1}^{|\Lambda|} (-1)^{i+1} \cdot \left(\sum_{B \in [\Lambda]^i} \frac{1}{\sum_{\lambda \in B} \lambda} \right).$$

Proof. We use the following short-hand:

$$X_B = \sum_{\lambda \in B} \lambda.$$

We rewrite our proof obligation as follows:

$$\mathbf{E}\big(\max(\Lambda)\big) = \sum_{i=1}^{|\Lambda|}(-1)^{i+1} \cdot \left(\sum_{B\in[\Lambda]^i} \frac{1}{X_B}\right) = \sum_{\substack{B\subseteq\Lambda\\ B\neq\emptyset}}(-1)^{|B|+1} \cdot \frac{1}{X_B}.$$

We perform an induction over the cardinality of Λ.

- $|\Lambda| = 1$: Then $\Lambda = \{\lambda\}$ and the expected time is

$$\frac{1}{\lambda} = \frac{1}{X_{\{\lambda\}}} = \sum_{\substack{B\subseteq\{\lambda\}\\ B\neq\emptyset}} (-1)^{|B|+1} \cdot \frac{1}{X_B}.$$

- $|\Lambda| = n$: We can decompose the expected time for n rates by considering the expected times after the first event happened:

$$\mathbf{E}\big(\max(\Lambda)\big) = \frac{1}{X_\Lambda} + \sum_{\lambda\in\Lambda} \frac{\lambda}{X_\Lambda} \cdot \mathbf{E}\big(\max(\Lambda \setminus \{\lambda\})\big).$$

  The formula can easily be seen by looking at the CTMC state space for an AND-gate over rates Λ.

  We can apply the induction hypothesis because $\Lambda \setminus \{\lambda\}$ has cardinality $n-1$.

$$\begin{aligned}
&\frac{1}{X_\Lambda} + \sum_{\lambda\in\Lambda} \frac{\lambda}{X_\Lambda} \cdot \mathbf{E}\big(\max(\Lambda \setminus \{\lambda\})\big)\\
=&\frac{1}{X_\Lambda} + \sum_{\lambda\in\Lambda} \frac{\lambda}{X_\Lambda} \cdot \left(\sum_{\substack{B\subseteq\Lambda\setminus\{\lambda\}\\ B\neq\emptyset}} (-1)^{|B|+1} \cdot \frac{1}{X_B}\right).
\end{aligned}$$

  This can be rewritten by using the insight that each multi-set $B \subseteq \Lambda\setminus\{\lambda\}$ appears exactly for those rates λ which are not contained in B.

$$
\begin{aligned}
&\frac{1}{X_\Lambda} + \sum_{\lambda \in \Lambda} \frac{\lambda}{X_\Lambda} \cdot \left(\sum_{\substack{B \subseteq \Lambda \setminus \{\lambda\} \\ B \neq \emptyset}} (-1)^{|B|+1} \cdot \frac{1}{X_B} \right) \\
=& \frac{1}{X_\Lambda} + \frac{1}{X_\Lambda} \cdot \left(\sum_{\substack{B \subseteq \Lambda \\ 0 < |B| < |\Lambda|}} \sum_{\lambda \notin B} \lambda \cdot (-1)^{|B|+1} \cdot \frac{1}{X_B} \right) \\
=& \frac{1}{X_\Lambda} + \frac{1}{X_\Lambda} \cdot \left(\sum_{\substack{B \subseteq \Lambda \\ 0 < |B| < |\Lambda|}} \left(X_\Lambda - X_\Lambda + \sum_{\lambda \notin B} \lambda \right) \cdot (-1)^{|B|+1} \cdot \frac{1}{X_B} \right) \\
=& \frac{1}{X_\Lambda} + \frac{1}{X_\Lambda} \cdot \left(\sum_{\substack{B \subseteq \Lambda \\ 0 < |B| < |\Lambda|}} (X_\Lambda - X_B) \cdot (-1)^{|B|+1} \cdot \frac{1}{X_B} \right) \\
=& \frac{1}{X_\Lambda} + \frac{1}{X_\Lambda} \cdot \left(\sum_{\substack{B \subseteq \Lambda \\ B \neq \emptyset}} (X_\Lambda - X_B) \cdot (-1)^{|B|+1} \cdot \frac{1}{X_B} \right) \\
=& \frac{1}{X_\Lambda} + \frac{1}{X_\Lambda} \cdot \left(\sum_{\substack{B \subseteq \Lambda \\ B \neq \emptyset}} (-1)^{|B|+1} \cdot \frac{X_\Lambda}{X_B} - \sum_{\substack{B \subseteq \Lambda \\ B \neq \emptyset}} (-1)^{|B|+1} \right) \\
=& \frac{1}{X_\Lambda} + \sum_{\substack{B \subseteq \Lambda \\ B \neq \emptyset}} (-1)^{|B|+1} \cdot \frac{1}{X_B} - \sum_{\substack{B \subseteq \Lambda \\ B \neq \emptyset}} (-1)^{|B|+1} \cdot \frac{1}{X_\Lambda} \\
=& \sum_{\substack{B \subseteq \Lambda \\ B \neq \emptyset}} (-1)^{|B|+1} \cdot \frac{1}{X_B} + \frac{1}{X_\Lambda} \left(1 - \sum_{\substack{B \subseteq \Lambda \\ B \neq \emptyset}} (-1)^{|B|+1} \right) \\
=& \sum_{\substack{B \subseteq \Lambda \\ B \neq \emptyset}} (-1)^{|B|+1} \cdot \frac{1}{X_B}.
\end{aligned}
$$

C

This successfully shows the proof obligation. The last step uses the equation

$$1 - \sum_{\substack{B \subseteq \Lambda \\ B \neq \emptyset}} (-1)^{|B|+1} = 1 + \sum_{\substack{B \subseteq \Lambda \\ B \neq \emptyset}} (-1)^{|B|} = 0.$$

This can be seen by using the fact that the number of subsets of a given cardinality is captured by the binomial numbers:

$$\sum_{\substack{B \subseteq \Lambda \\ B \neq \emptyset}} (-1)^{|B|} = \sum_{i=1}^{|\Lambda|} (-1)^i \cdot \binom{|\Lambda|}{i} = -1 + \sum_{i=0}^{|\Lambda|} (-1)^i \cdot \binom{|\Lambda|}{i} = -1.$$

The last step follows from the fact that the alternating sum of binomial coefficients is zero. ■

D Additional information for evaluation

Artefact | The tools, example files and log files used in the evaluation are available in an artefact on Zenodo[1].

FFORT | We use the git repository of FFORT[2] with commit `f568269`. We consider the following benchmarks (369 in total):

- AHRS: 2 variants (13 benchmarks)
- CAS: 2 variants (4 benchmarks)
- CSD: 1 variant (2 benchmarks)
- DPHS: 1 variant (2 benchmarks)
- EMA: 1 variant (4 benchmarks)
- FTPP: 8 variants (16 benchmarks)
- HECS: 18 variants (37 benchmarks)
- HEMPS: 1 variant (2 benchmarks)
- MAS: 2 variants (4 benchmarks)
- MCS: 1 variant (7 benchmarks)
- MPS: 1 variant (2 benchmarks)
- NPPW: 1 variant (6 benchmarks)
- OGPF: 1 variant (2 benchmarks)
- PCBA: 1 variant (2 benchmarks)
- PCS: 1 variant (2 benchmarks)
- PT: 1 variant (2 benchmarks)
- RC: 68 variants (136 benchmarks)
- SF: 39 variants (78 benchmarks)
- SMS: 12 variants (24 benchmarks)
- ST: 3 variants (6 benchmarks)
- VGS: 8 variants (16 benchmarks)
- WQDN: 1 variant (2 benchmarks)

Tools | We use Storm-dft in version 1.6.3 (with commit `c12c035`). DFTCalc with commit `529c7fb` uses DFTRES as back-end and Storm for model checking. We

[1] https://doi.org/10.5281/zenodo.7834056
[2] https://dftbenchmarks.utwente.nl/public/ffort.git

D

use DFTRES with commit `e5895c9`. For each tool and configuration we give the configuration arguments:

- Storm-dft (plain): `--disabledc -nosymred`
- Storm-dft (DC): `-nosymred`
- Storm-dft (Symred): default arguments
- Storm-dft (Mod): `--modularisation`
- Storm-approx (Depth):
  `-approx 1e-4 --approximationheuristic depth`
- Storm-approx (Prob):
  `-approx 1e-4 --approximationheuristic probability`
- Storm-approx (Bound):
  `-approx 1e-4 --approximationheuristic bounddifference`
- DFTCalc: `--exact --storm`
- DFTRES: `--relErr 1e-4 -p 1`

E List of Symbols

Analysis of DFTs

I_v^t	Birnbaum importance index of v within time t 196
$\mathbf{MTTF}_v$	Mean-time-to-failure (MTTF) of component v 195
$\mathbf{OperationalBEs}(\mathbf{state}(\tau))$	Operational BEs for $\mathsf{state}(\tau)$ 224
$\mathbf{PropDep}(\tau)$	Propagation (without dependencies) for trace τ 185
$\mathbf{Reliability}_v^t$	Reliability of component v within t time units 195
$\mathbf{Unreliability}_v^t$	Unreliability of component v within t time units 195
$\mathcal{M}_{\mathcal{F}}$	Markov automaton for DFT $\mathcal{F}$ 187
$\mathbf{state}(\tau)_v$	State of element v after trace τ 181
$\mathbf{state}(\tau)$	DFT state after trace τ 181
$\mathbf{sym}_{s_2}^{s_1}$	Symmetry relation between subtrees of s_1 and s_2 203

DFT to GSPN Translation

$\mathcal{G}_{\mathcal{F}}$	GSPN template for DFT $\mathcal{F}$ 125
$\mathcal{I}_{\mathcal{F}}$	Interface places 58
$\boldsymbol{Active}_v$	Active place 62
$\boldsymbol{Disabled}_v$	Disabled place 120
$\boldsymbol{Failed}_v$	Failure place 61
$\boldsymbol{Unavail}_v$	Unavailable place 97
$\mathbf{m}(\tau)$	Marking after trace τ 61
$\mathbf{merge}(\mathcal{T}_1, \mathcal{T}_2)$	Merging of GSPN templates 59
$\mathbf{templ}_t(v)$	GSPN template for DFT element v of type t 61
$\mathcal{T}$	GSPN template 58

DFT Traces

$\mathbf{Eval}$	Evaluation of element v after trace τ 33
$\mathbf{Events}_{\mathcal{F}}^{\blacktriangleright}$	Set of event traces in DFT $\mathcal{F}$ 31
$\mathbf{Events}_{\mathcal{F}}$	Set of DFT events in DFT $\mathcal{F}$ 31
$\mathbf{Events}^v$	Events of DFT element v 34
$\mathbf{Fails}_{\mathbf{SE}_{\mathcal{F}}}^{\blacktriangleright}$	Set of failure traces in DFT $\mathcal{F}$ 32
$\mathbf{NextEval}^{Sem}$	Next elements to evaluate 37
$\mathbf{NextEvent}^{Sem}(\tau)$	Next events after trace τ according to semantics Sem 37
$\mathbf{Traces}^{Sem}(\boldsymbol{\tau}_{\mathrm{f}})$	Valid traces for ftrace $\boldsymbol{\tau}_{\mathrm{f}}$ according to semantics Sem 38

$\uparrow_v$	Activation of DFT node v 31
cl_v^s	Claiming of child v by SPARE s 31
f_v	Failure of DFT node v 31
$\boldsymbol{\tau}_{\mathbf{f}}$	Failure trace 32
$\varnothing_{\mathcal{F}}$	Set of no events in DFT $\mathcal{F}$ 31
$\oslash_v$	No event for DFT node v 31
$\mathbf{Fails}_{\mathbf{SE}_{\mathcal{F}}}$	Set of SE failures in DFT $\mathcal{F}$ 31
$\mathbf{f}_v$	Failure of SE v by itself 31
τ	Event trace 31
$\blacktriangleleft_v^\tau$	Observed trace order for DFT element v and trace τ 34
$\lightning_v$	Violation of SEQ v 31

Dynamic Fault Tree (DFT)

Elements	Types of DFT elements 22
Gates	Types of DFT gates 22
Leaves	Types of DFT leaves 22
SE	Stochastic event 22
$\mathbf{SE}_{exp}$	Stochastic event with exponential distribution 23
Θ	Attachment function for SEs 23
σ	Children of a DFT node 22
$\mathcal{F}^\Omega$	DFT over probability distributions Ω 22
$E(\sigma)$	Edges of DFT $\mathcal{F}$ 23
$G(\mathcal{F})$	Graph of DFT $\mathcal{F}$ 23
$\mathbf{subtree}(\mathcal{F}, v)$	Subtree of DFT element v in DFT $\mathcal{F}$ 24
$\mathbf{subtreeExtended}(\mathcal{F}, v)$	Extended subtree of DFT element v in DFT $\mathcal{F}$ 24
FD_a	Active failure distribution 23
FD_p	Passive failure distribution 23
d	Dormancy factor 26
$\mathbf{BE}_{\mathcal{F}}$	Set of basic events 23
V	Nodes of a DFT 22
$\rho(v)$	Parents of a DFT node 23
λ_a	Active failure rate 26
λ_p	Passive failure rate 26
top	Top level event of a DFT 22
Type	Type of a DFT node 22

Generalized Stochastic Petri Net (GSPN)

M	Markings 47
$\mathbf{conc}(m)$	Transitions which have concession in marking m 49
$\mathbf{enabled}(m)$	Enabled transitions in marking m 49
$\mathcal{G}$	Generalized Stochastic Petri Net (GSPN) 48

m_0	Initial marking 47
$\mathcal{N}$	Petri net 47
$\mathcal{D}$	Partitioning of immediate transitions 48
P	Places 47
Π	Priority function 48
$\mathbf{RG}(\mathcal{G})$	Reachability graph of GSPN $\mathcal{G}$ 51
T	Transitions 47
W	Weight function 48

Markov automaton (MA)

AP	Atomic propositions 42
C	Contiuous-time Markov chain (CTMC) 44
$\mathbf{rate}(s)$	Exit rate of state s 43
s_0	Initial state 42
L	State labelling 42
$\mathcal{M}$	Markov automaton 42
Act	Actions 42
$\dashrightarrow$	Markovian transitions 42
$\hookrightarrow$	Probabilistic transitions 42
$\mathbf{MT}(s)$	Markovian transitions of s 42
$\mathbf{PT}(s)$	Probabilistic transitions of s 42
$\mathbf{rate}(s, s')$	Rate between states s and s' 43
S	States 42

Probability distribution

$\Omega_{\boldsymbol{exp}}$	Exponential distributions 20
Ω	Set of probability distributions 18
μ	Discrete probability distribution 18
Distr(S)	Discrete probability distributions over S 18
ω	Probability distribution 18
λ	Rate of exponential distribution 19

Sequence

ε	Empty sequence 17
ξ	Sequence 17
$\lvert\xi\rvert$	Size of sequence ξ 17

State of a DFT

$\mathbf{Active}(\tau)$	Active elements after trace τ 41
$\mathbf{Available}(\tau)$	Available for claiming after trace τ 91
$\mathbf{Claimed}(\tau)(s)$	Claimed elements of SPARE s after trace τ 41
$\mathbf{CurrentlyClaiming}(\tau)$	Currently claiming SPAREs after trace τ 102

E

$\textbf{Disabled}(\tau)$	Disabled elements after trace τ 41
$\textbf{Failed}(\tau)$	Failed elements after trace τ 40
$\textbf{NextSpare}(\tau, v)$	Next spare child to claim for SPARE v after trace τ 92

F Index

F

F

F